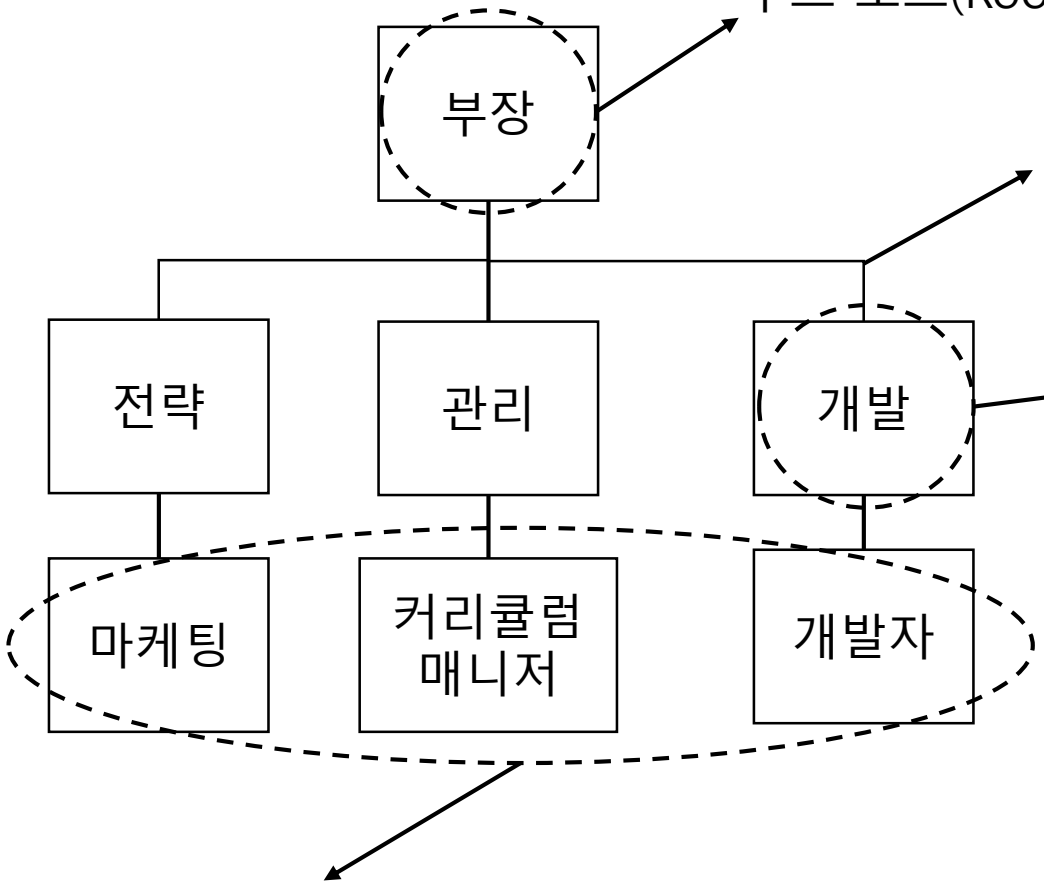
# 자료구조

1108 서민준

# 트리 (1)




루트 노드(Root Node) : 트리 구조에서 최상위에 있는 노드

간선(Edge) : 노드와 노드를 연결하는 연결선

노드(Node) : 트리의 구성요소에 해당합니다.
**노드 자체만으로도 트리**라고 할 수 있습니다.

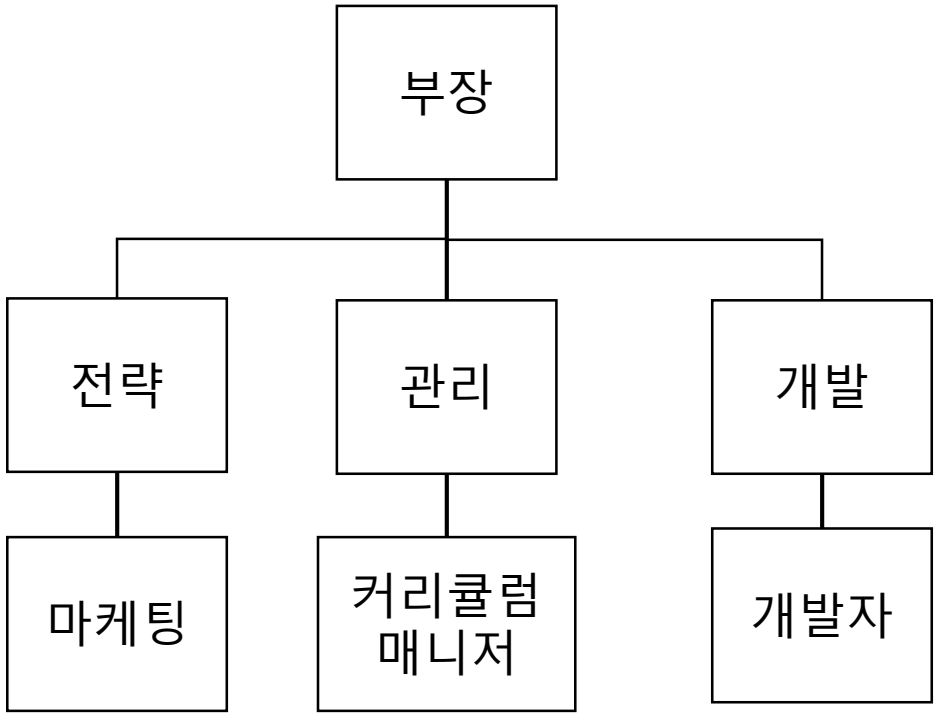
비단말 노드 : 단말노드를 제외한 모든 노드

단말 노드(Terminal Node) : 아래로 또 다른 노드가 연결되어 있지 않는 노드

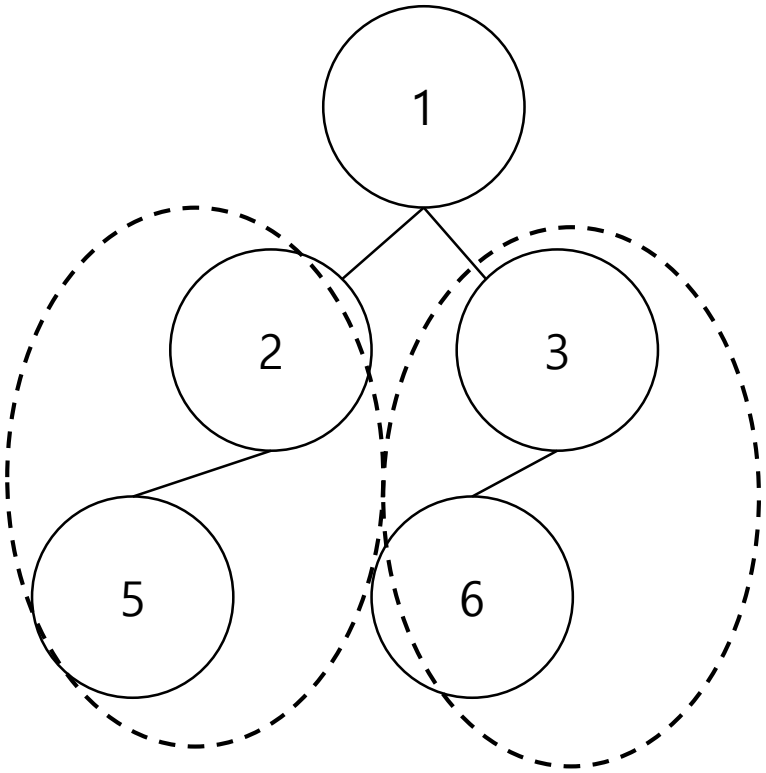
# 트리 (2)

차수(Degree) : 자식 노드의 수

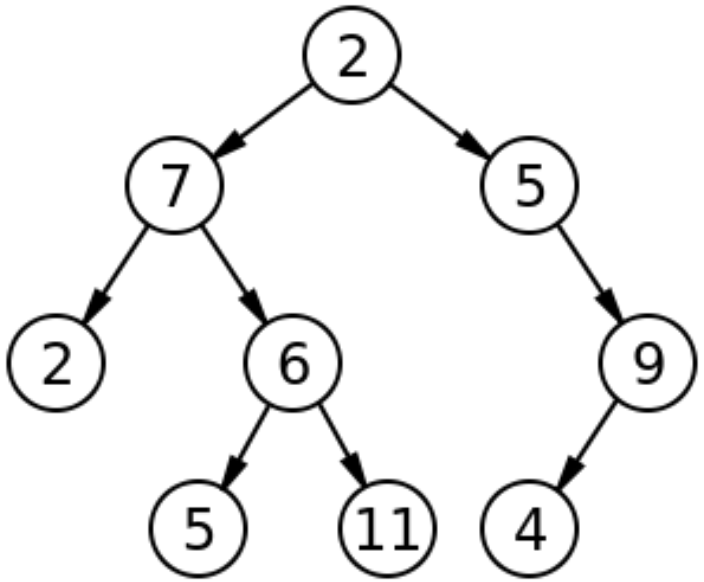
높이 : 노드의 차수 중 가장 큰 차수




서브 트리(Sub Tree)



# 이진 트리 (1)



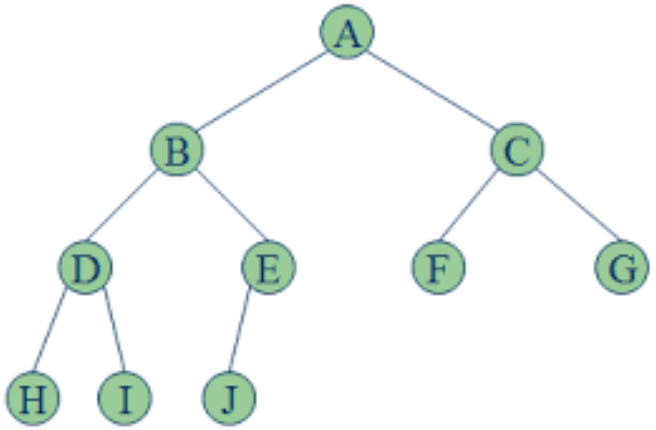

정의

**자식 노드**를 **최대 2개**까지 가질 수 있고, 자식 노드는 **공집합(NULL)**일 수도 있는 트리

조건

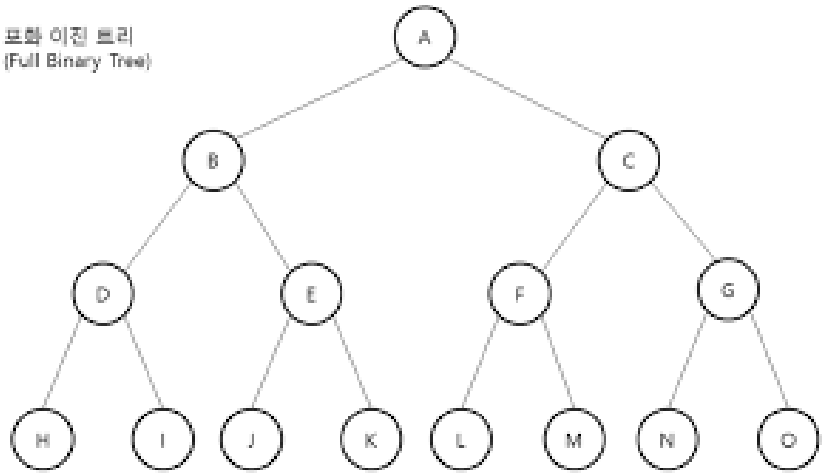
1. 루트 노드를 중심으로 두 개의 서브 트리로 나뉘어진다.
2. 나뉘어진 두 서브 트리도 모두 이진 트리이여야 한다.

# 이진 트리 (2)




완전 이진 트리

**위에서 아래로, 왼쪽에서 오른쪽**으로 **빈틈없이** 노드가 채워져 있는 상태의 이진 트리

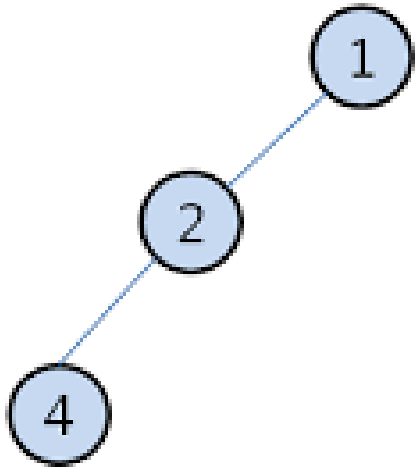



포화 이진 트리

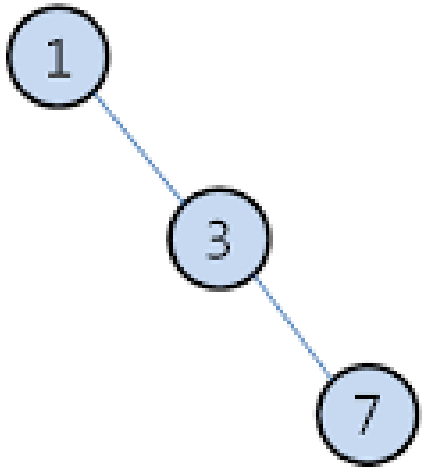
**완전 이진 트리** 상태 중 하나로, **모든 레벨**에 노드가 다 채워져 있는 트리

레벨의 최대 노드 수 : $2^{level}$

# 이진 트리 (3)



왼쪽편향 이진트리



오른쪽편향 이진트리

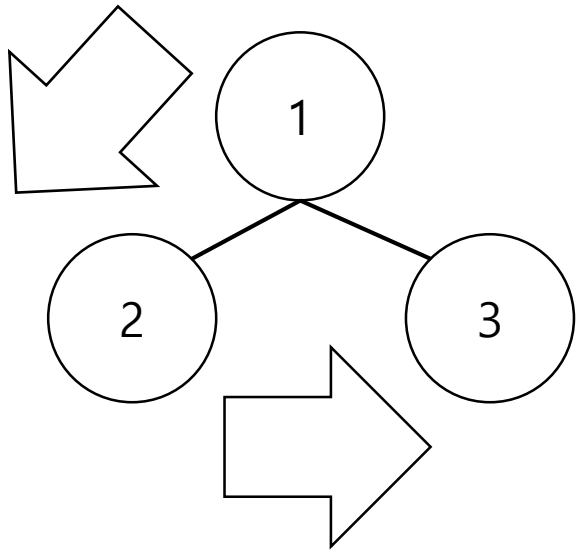
편향 이진 트리

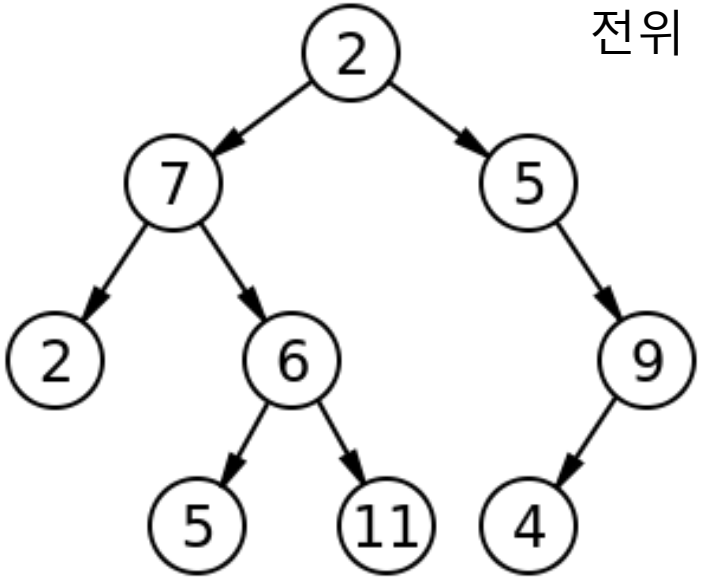
노드가 **왼쪽 혹은 오른쪽으로만** 연결되어 있는 이진 트리

# 이진 트리 순회 (1)

순회 : 트리에 존재하는 모든 노드를 도는 방법

**1. 전위 순회 : 루트 노드**를 **가장 먼저** 돈다.

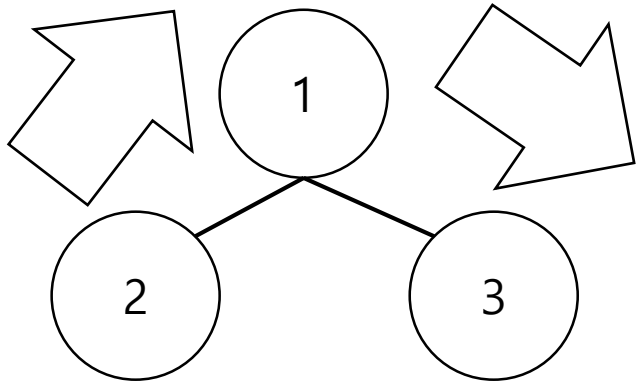


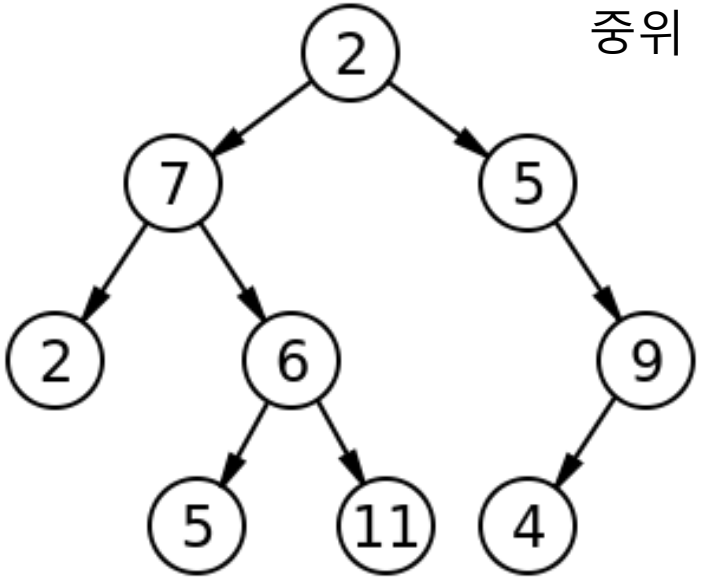




전위 순회 결과는?

# 이진 트리 순회 (2)

**2. 중위 순회 : 루트 노드**를 **2번째로** 돈다.

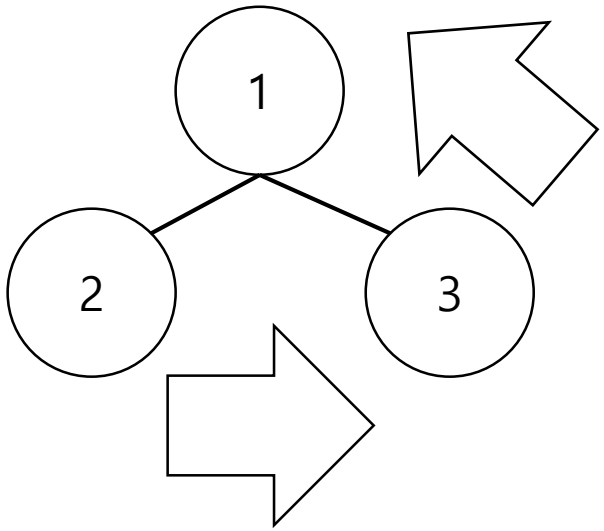


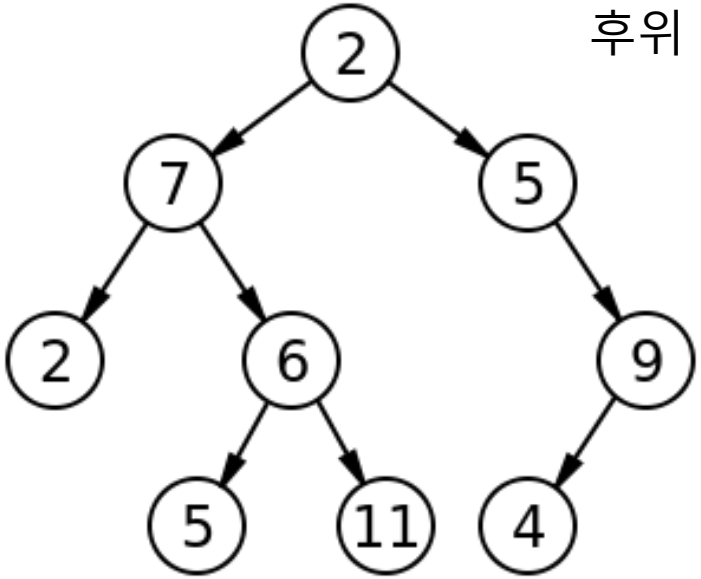




중위 순회 결과는?

# 이진 트리 순회 (3)

**3. 후위 순회 : 루트 노드**를 **가장 마지막**에 돈다.



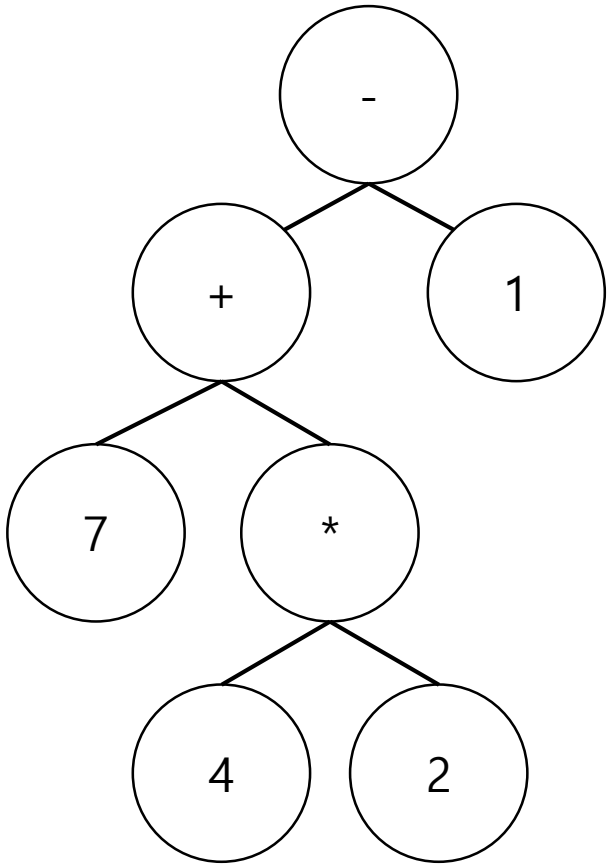




후위 순회 결과는?

# 수식 트리 (1)

중위 표현식
7 + 4 * 2 – 1

트리화

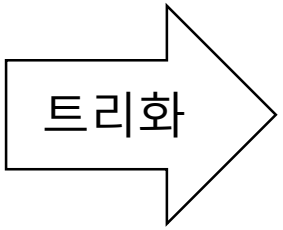


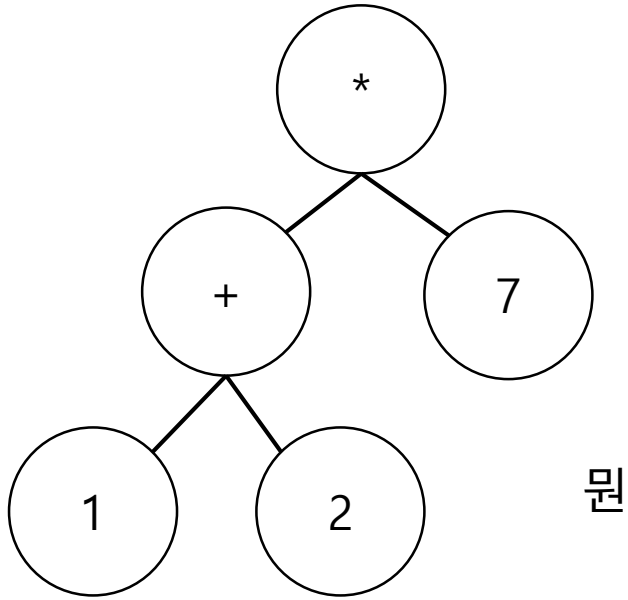

대체 어떤 규칙으로 생성되었는가?

# 수식 트리 (2)

중위 표현식으로 표현된 것을 바로 트리로 만드는 것보다
후위 표현식으로 표현된 식을 트리로 표현하는 것이 더 쉽다.

후위 표현식
1 2 + 7 *

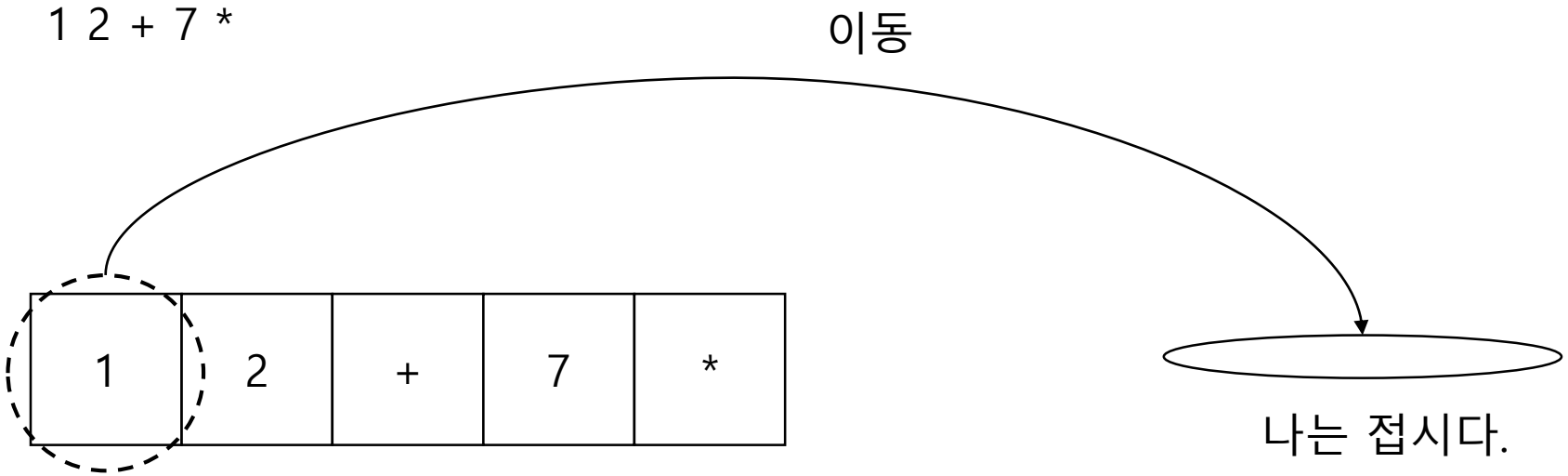


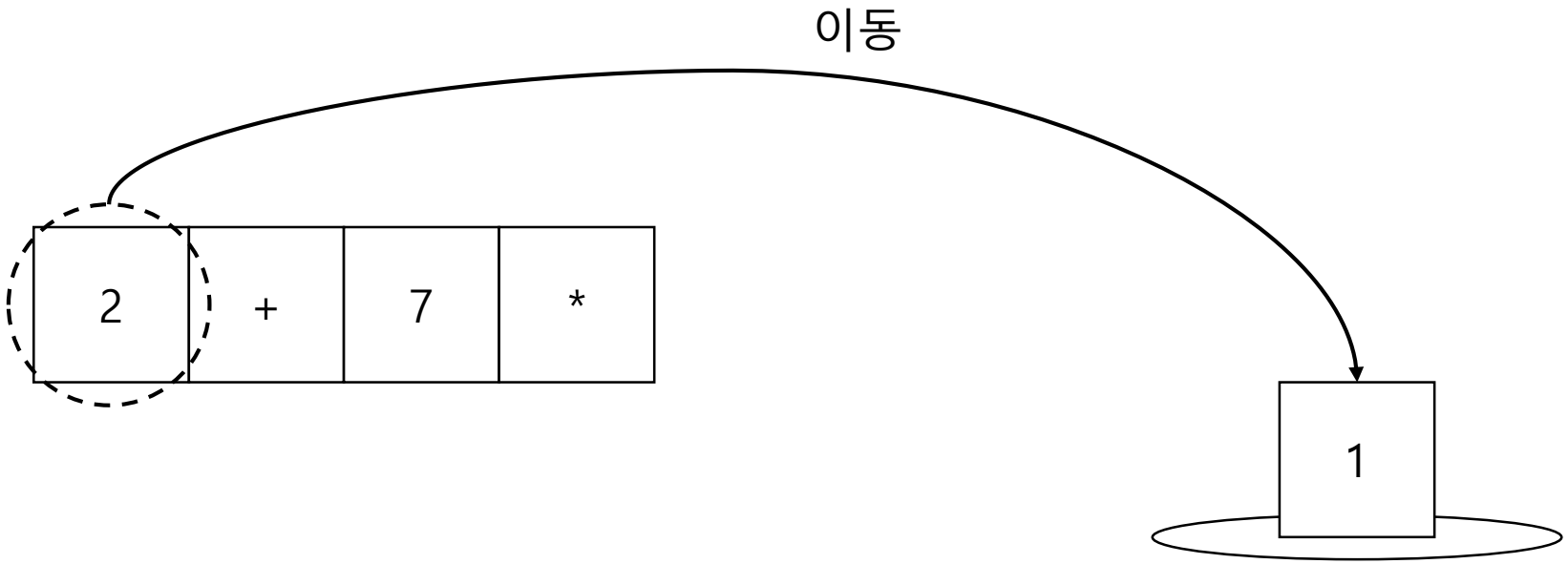

뭔가 규칙이 보이지 않는가?

# 수식 트리 (3)

후위 표현식
1 2 + 7 *



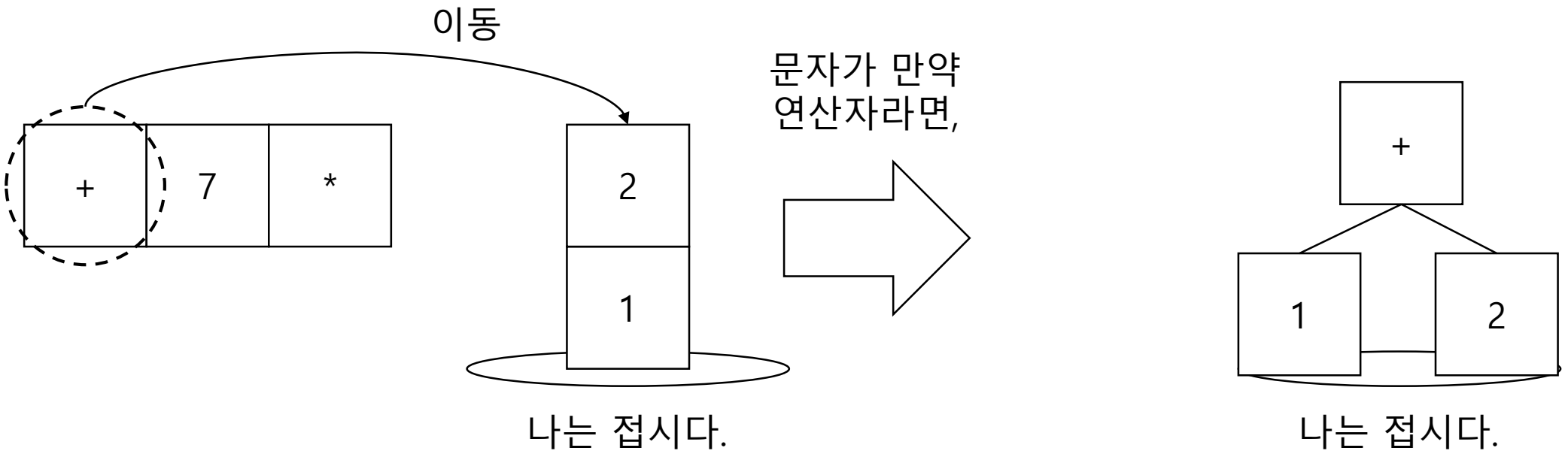
# 수식 트리 (4)



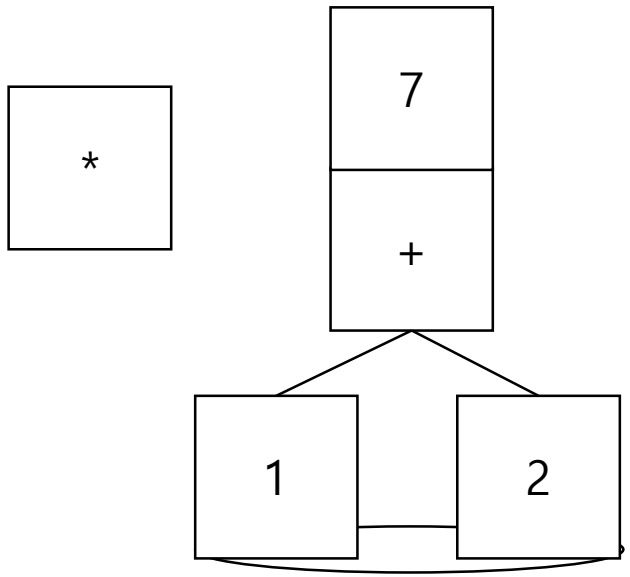

피연산자는 무조건 접시(스택)으로 옮긴다.

나는 접시다.

# 수식 트리 (5)

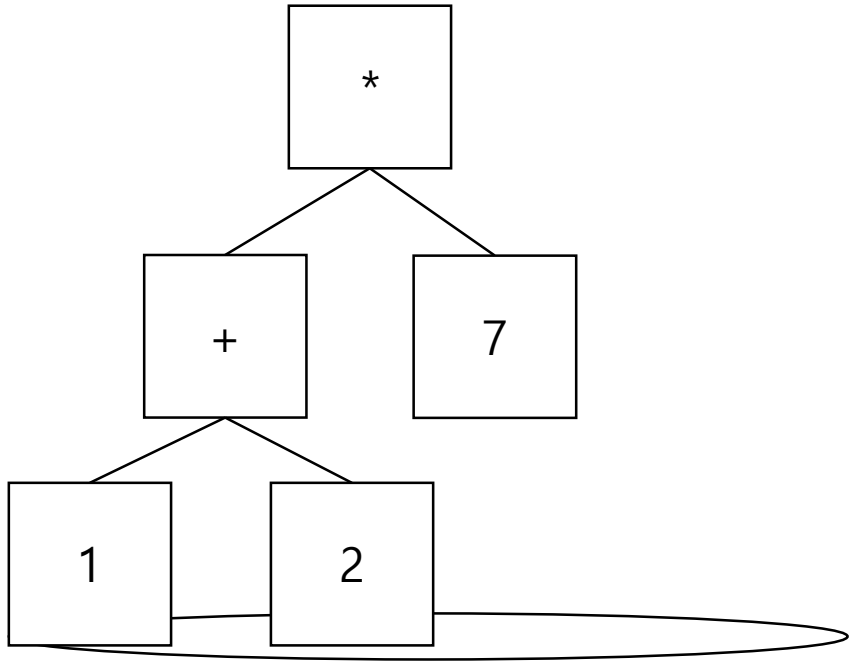


# 수식 트리 (6)




나는 접시다.

최종




나는 접시다.

# 수식 트리 (7)

수식 트리를 만드는 과정 정리!

- 피연산자를 만나면 무조건 접시(스택)으로 옮긴다.
- 연산자를 만나면 스택에서 두 개의 피연산자를 꺼내어 자식 노드로 연결
- 자식 노드를 연결해서 만들어진 트리는 다시 접시(스택)으로 옮긴다.

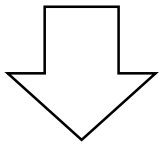
순회 방법에 따라 표기되는 식이 달라짐

전위 순회 시 전위 표기법
중위 순회 시 중위 표기법
후위 순회 시 후위 표기법

# 수식 트리 (8)

중위 식으로 표기할 때 소괄호를 출력하는 방법

후위 표현식
1 2 + 7 *



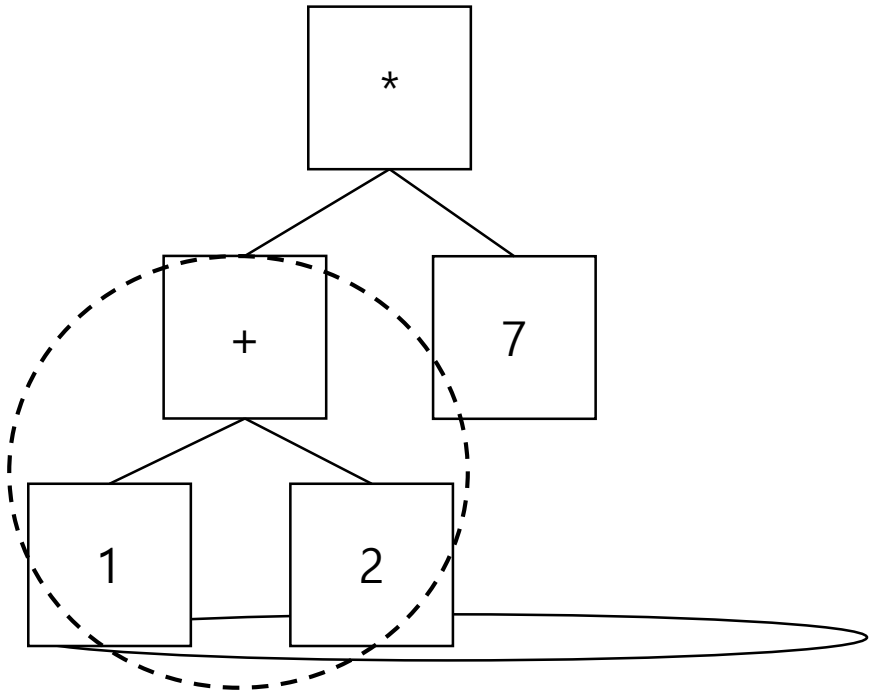
중위 표현식
( ( 1 + 2 ) * 7 )

↳ 연산자의 수와 소괄호의 수가 일치한다!

```
void ShowInfixTypeExp(BTreeNode* bt)
{
    if (bt == NULL)
        return;

    if (bt->left != NULL || bt->right != NULL)
        fputs("(", stdout);

    ShowInfixTypeExp(bt->left);
    printf("%c ", bt->data);
    ShowInfixTypeExp(bt->right);

    if (bt->left != NULL || bt->right != NULL)
        fputs(")", stdout);
}
```

# 수식 트리 (9)

그럼 이제 만들어진 트리를 계산해보자.

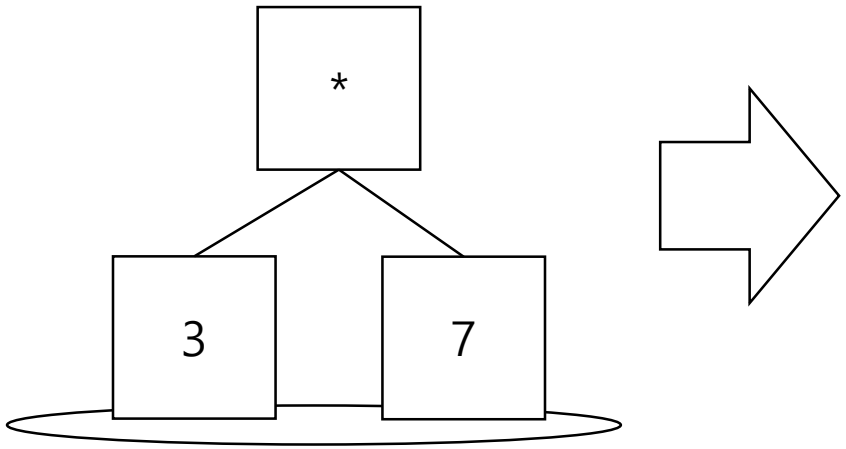



나는 접시다.

1. 가장 아래에 있는 1과 2를 + 연산을 하고 그 값을 반환한다.
   후위 순회로 돌면 1 2 +가 된다.

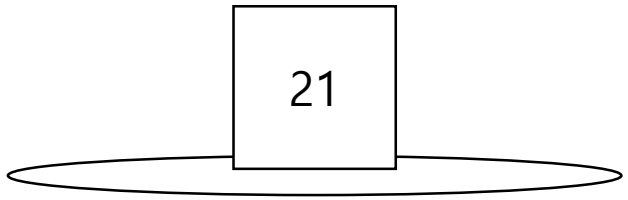
# 수식 트리 (10)

2. 다시 후위 순회를 하여 연산을 진행한다.




나는 접시다.

3. 남은 것이 루트 노드 하나이므로 루트 노드를 출력한다.




나는 접시다.

# 수식 트리 (11)

```
int EvaluateExpTree(BTreeNode* bt)
{
    int op1, op2;

    if (bt->left == NULL && bt->right == NULL) // 단말 노드라면
        return bt->data;

    // 피연산자를 구한다.
    op1 = EvaluateExpTree(bt->left);
    op2 = EvaluateExpTree(bt->right);

    switch (bt->data) // 연산자의 종류별로 연산 진행 후 반환
    {
    case '+':
        return op1 + op2;
    case '-':
        return op1 - op2;
    case '*':
        return op1 * op2;
    case '/':
        return op1 / op2;
    }
    return 0;
}
```

# 우선순위 큐 (1)

일반적인 큐와 동일한 연산을 가진다.

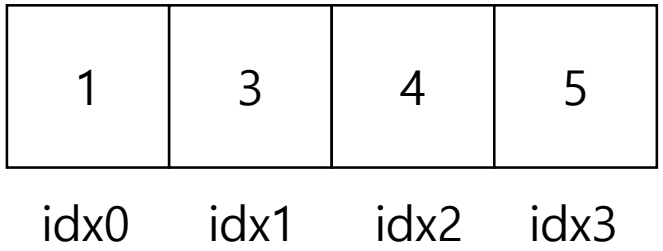
enqueue : 우선순위 큐에 데이터를 삽입하는 행위
dequeue : 우선순위 큐에서 데이터를 꺼내는 행위

단, 연산의 결과에서 차이가 있다.

우선순위 큐는 **들어간 순서에 상관없이, 우선순위가 높은 데이터가 먼저 나온다.**

# 우선순위 큐 (2)

배열로 구현하는 우선순위 큐



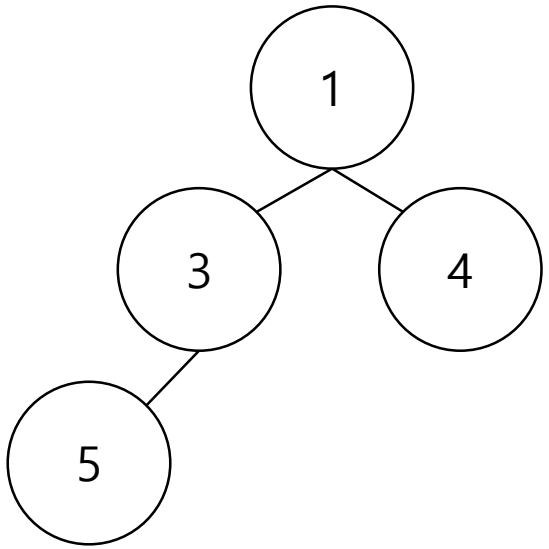

연결 리스트 기반의 우선순위 큐


단점

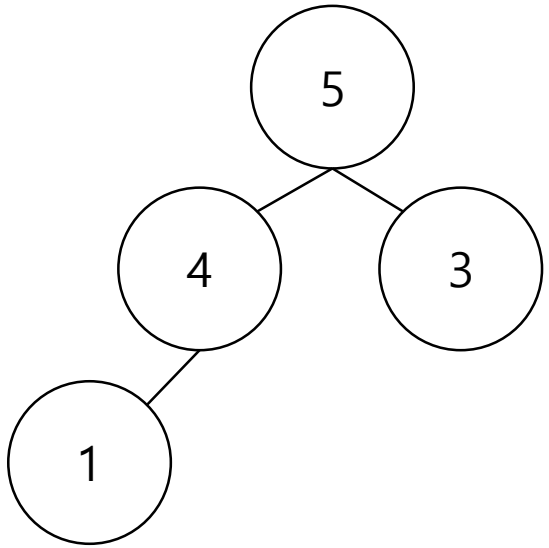
데이터를 삽입 / 삭제 과정에서 데이터를 shift하는 연산을 수반해야 한다.
삽입의 위치를 찾기 위해서 배열에 저장된 모든 데이터와 우선순위 비교를 해야 할 수도 있다.

# 우선순위 큐 (3)

힙으로 구현하는 우선순위 큐




최소 힙




최소 힙

# 힙 (1)

완전 이진 트리의 형태

모든 노드가 **우선순위에 따라 정렬**되어 있는 상태의 **이진 트리**

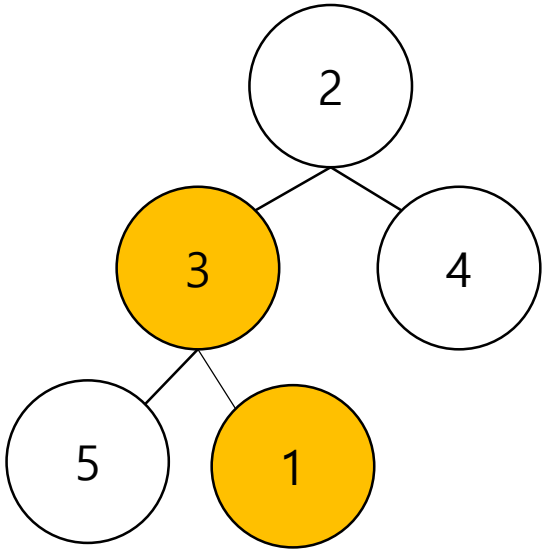
부모 노드가 자식 노드보다 **작거나 같으면 최소 힙**

부모 노드가 자식 노드보다 **크거나 같으면 최대 힙**

# 힙 (2)

여기서는 최소 힙(min heap)을 기준으로 설명합니다.

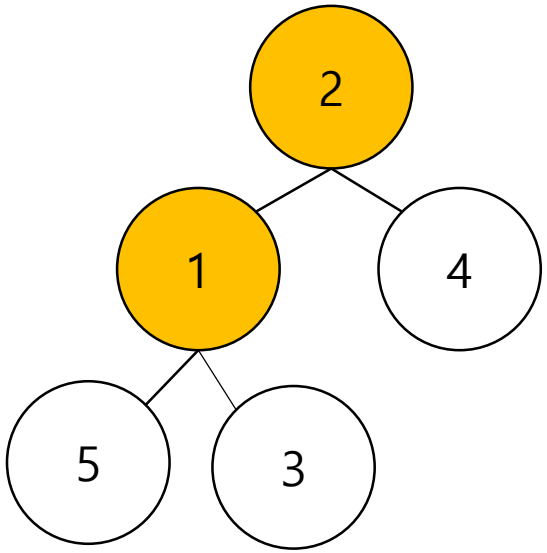
새로운 데이터는 우선순위가 제일 낮다는 가정하에 **'마지막 위치'** 에 저장한다.



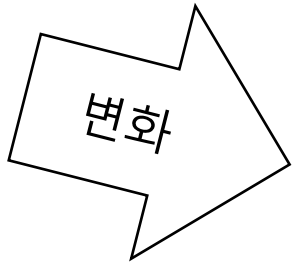

데이터를 삽입하였다면 자신의 부모 노드와 우선순위를 비교한다.
-> 새로 삽입된 자료가 우선순위가 더 높다.

만약 새로 삽입된 자료의 우선순위가 높다면 부모 노드의 값과 자식 노드의 값을 변경한다.

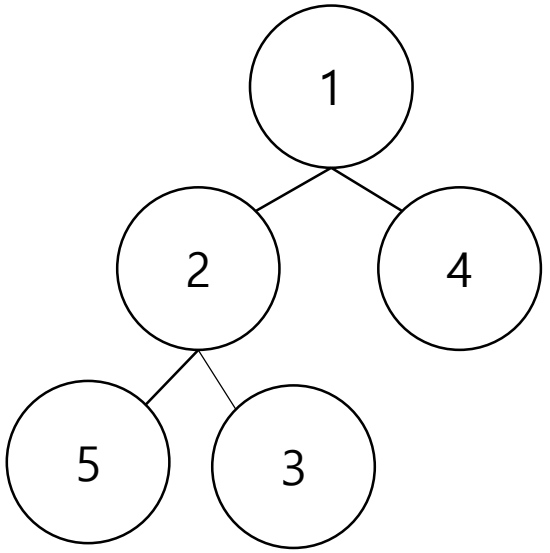
# 힙 (3)




아직 부모 노드가 존재하므로 부모 노드와 다시 한 번 우선순위를 비교한다.
-> 이번에도 우선순위가 자식 노드가 더 높다.

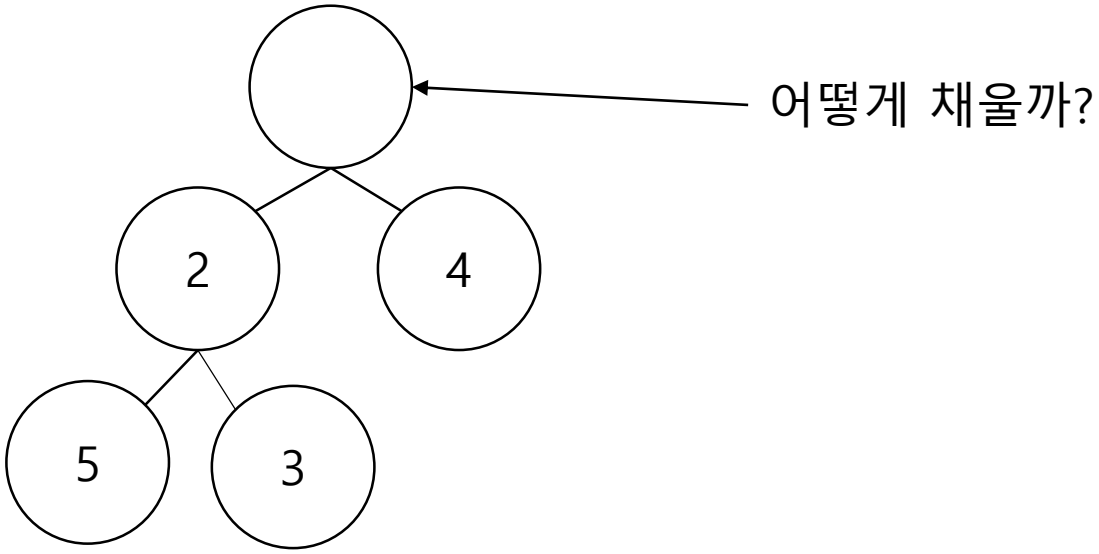



최종결과

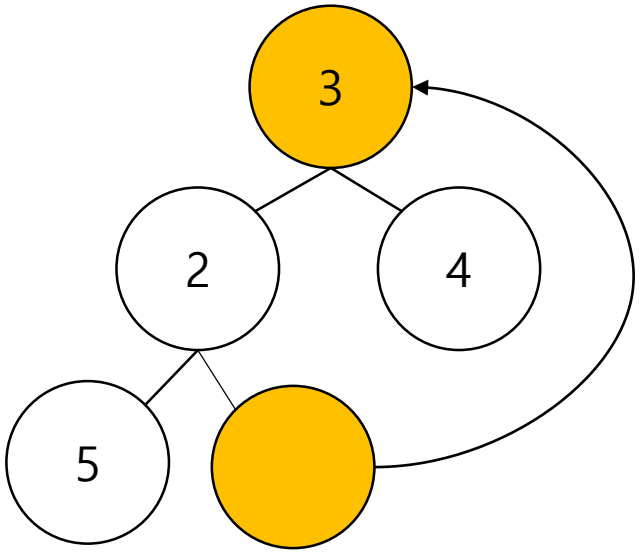


# 힙 (4)

이번에는 데이터를 삭제하는 과정을 알아보자.

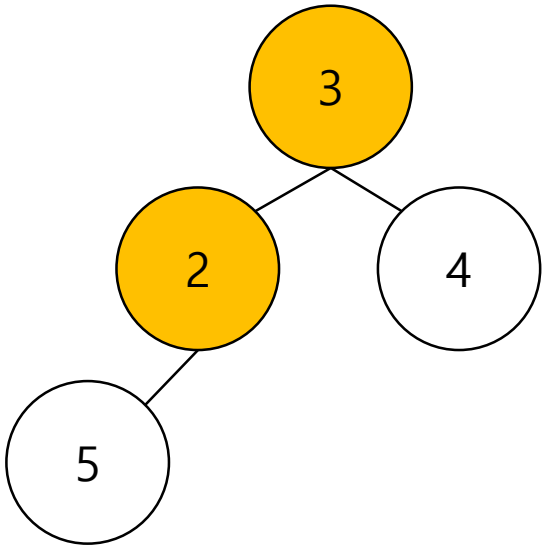


# 힙 (5)



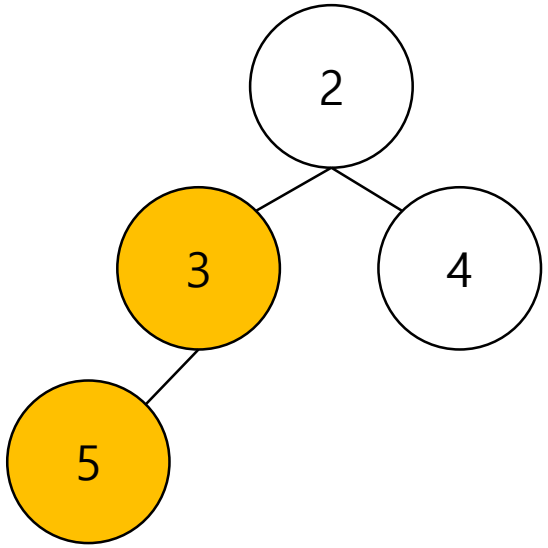

우선 가장 마지막에 있는 노드를 루트 노드로 옮긴다.

# 힙 (6)




그 뒤, 자식 노드와 우선순위를 비교하여 자신의 위치를 찾는다.
-> 현재 자식의 우선순위가 더 높으므로 위치 변경이 필요하다.

# 힙 (7)

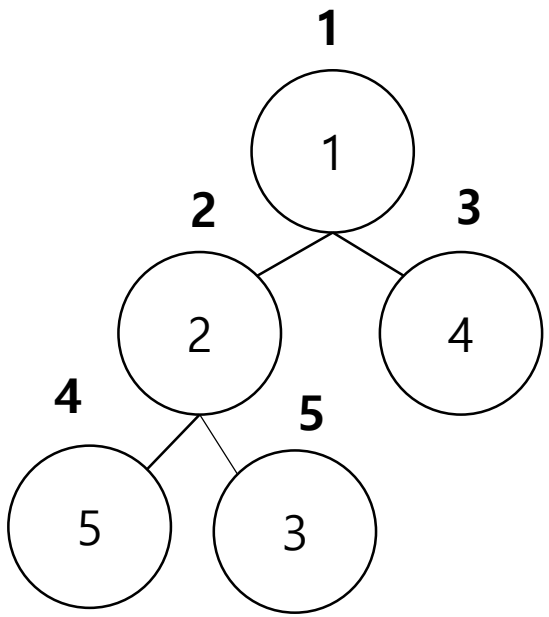



위치 변경을 한 후에도 자식 노드가 존재하므로 다시 한 번 우선순위를 검사한다.
-> 이번에는 자식 노드가 우선순위가 더 낮으므로 자리를 찾음

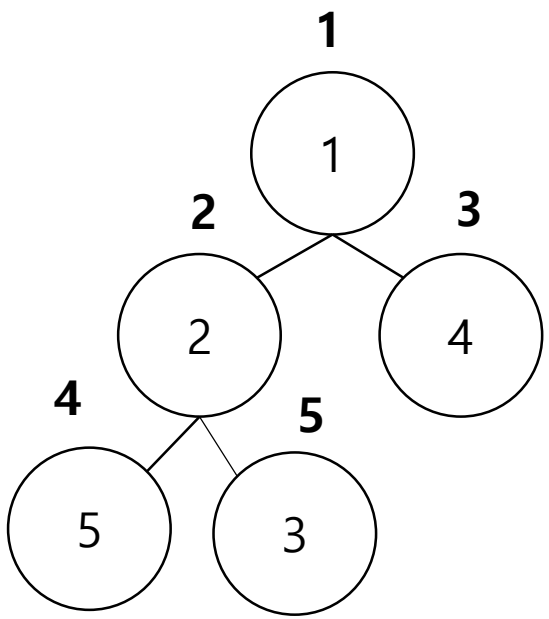
# 힙(8)

힙은 연결리스트로 구현하기보다는 **배열로 구현**하는 것이 더욱 **효과적**이다.
연결리스트로 구현하면 새로운 노드를 마지막에 추가하기가 어렵기 때문

배열로 힙을 구현할 때는 노드에 index를 가지고 접근을 하는데 방법은 다음과 같다.



# 힙 (9)




접근 인덱스를 1부터 시작하는데 이유는 다음과 같다.

0으로 시작해도 구할 수는 있다.
그러나 1로 시작하면 메모리 공간 1개를 낭비하는 대신에
**더 적은 연산으로 많은 이익**을 얻을 수 있다.

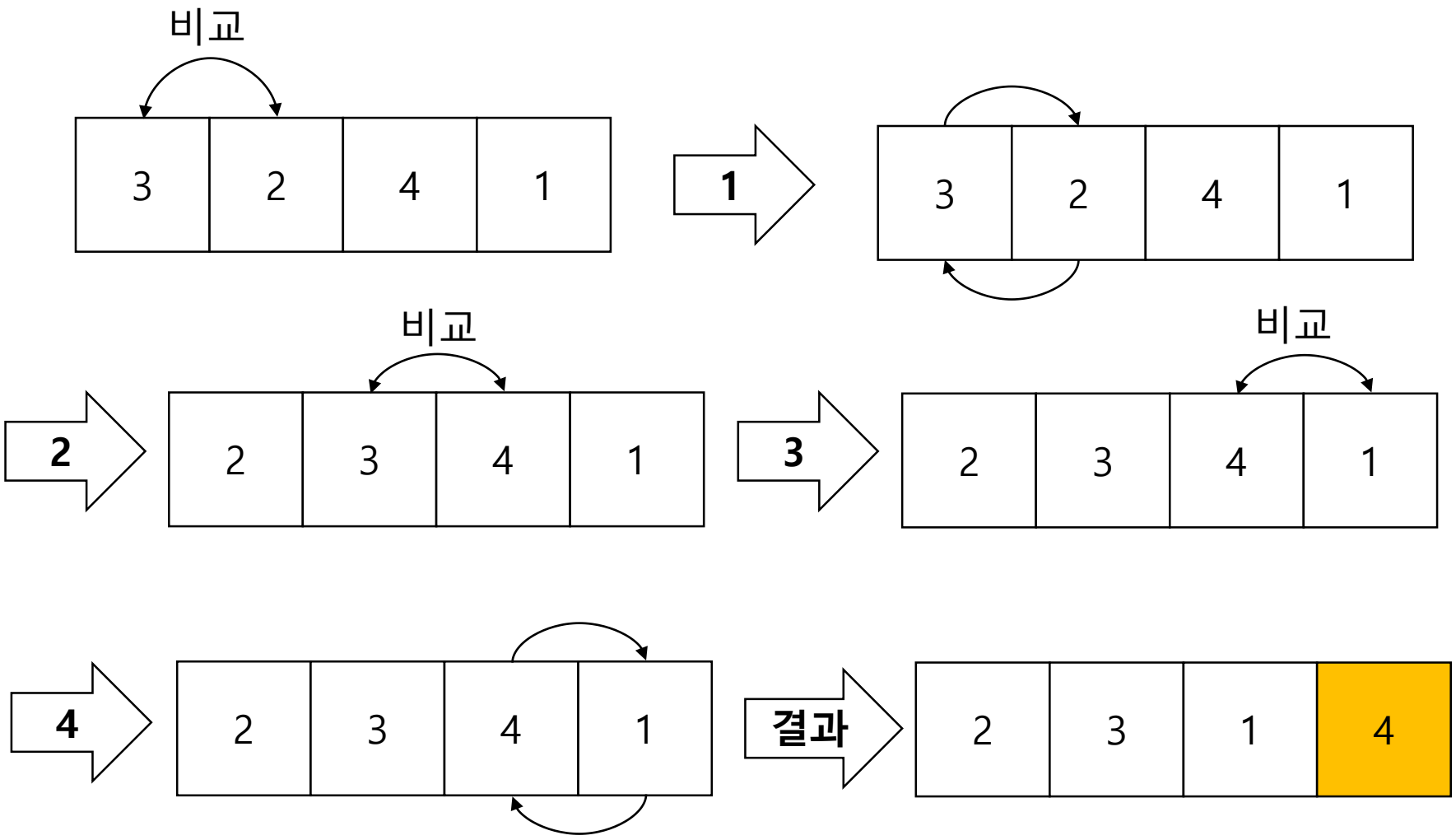
그리고 인덱스로 접근하므로 부모, 왼쪽 자식, 오른쪽 자식을 구할 수 있다.

부모 노드 인덱스 : 자식 노드의 인덱스 값 / 2

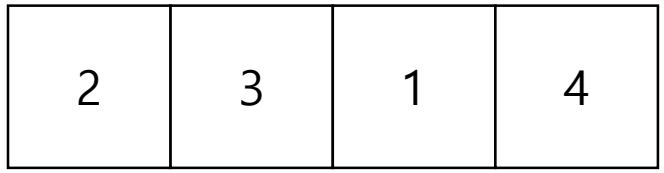
왼쪽 자식 인덱스 : 부모 노드의 인덱스 값 * 2

오른쪽 자식 인덱스 : 부모 노드의 인덱스 값 * 2 + 1

# 버블 정렬 (1)



# 버블 정렬 (2)



# 버블정렬 (3)

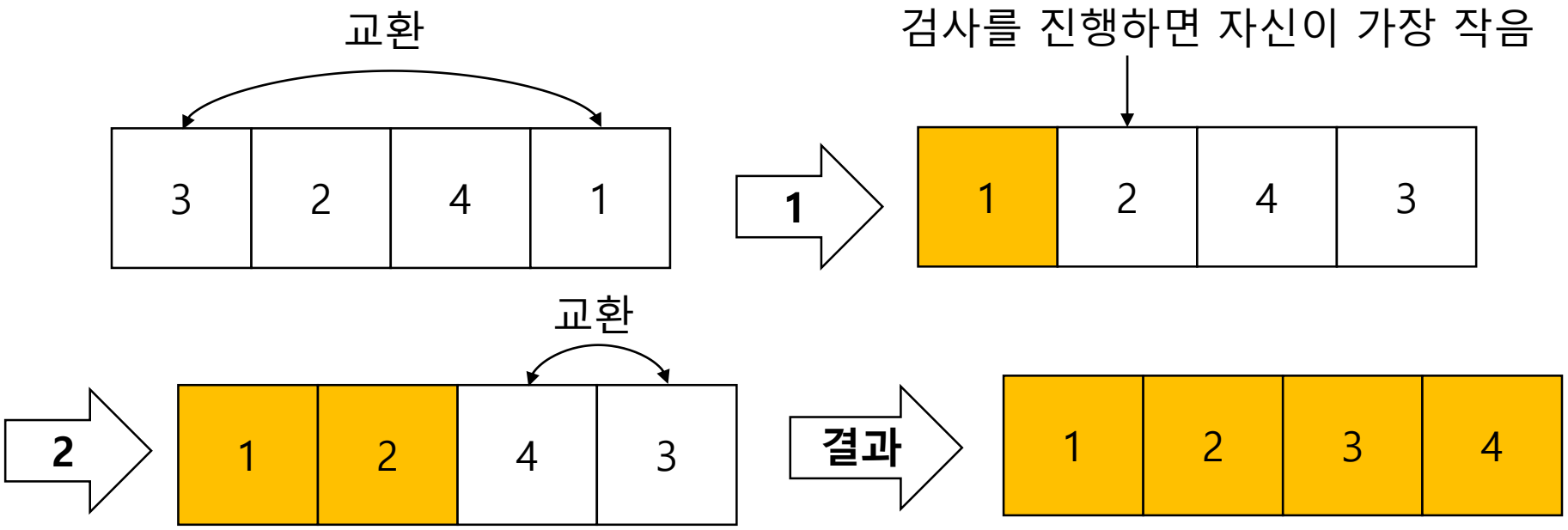
기존의 버블정렬은 이미 정렬이 다 된 상태에서도 검사를 진행한다.

| 1 | 2 | 3 | 4 |
|---|---|---|---|

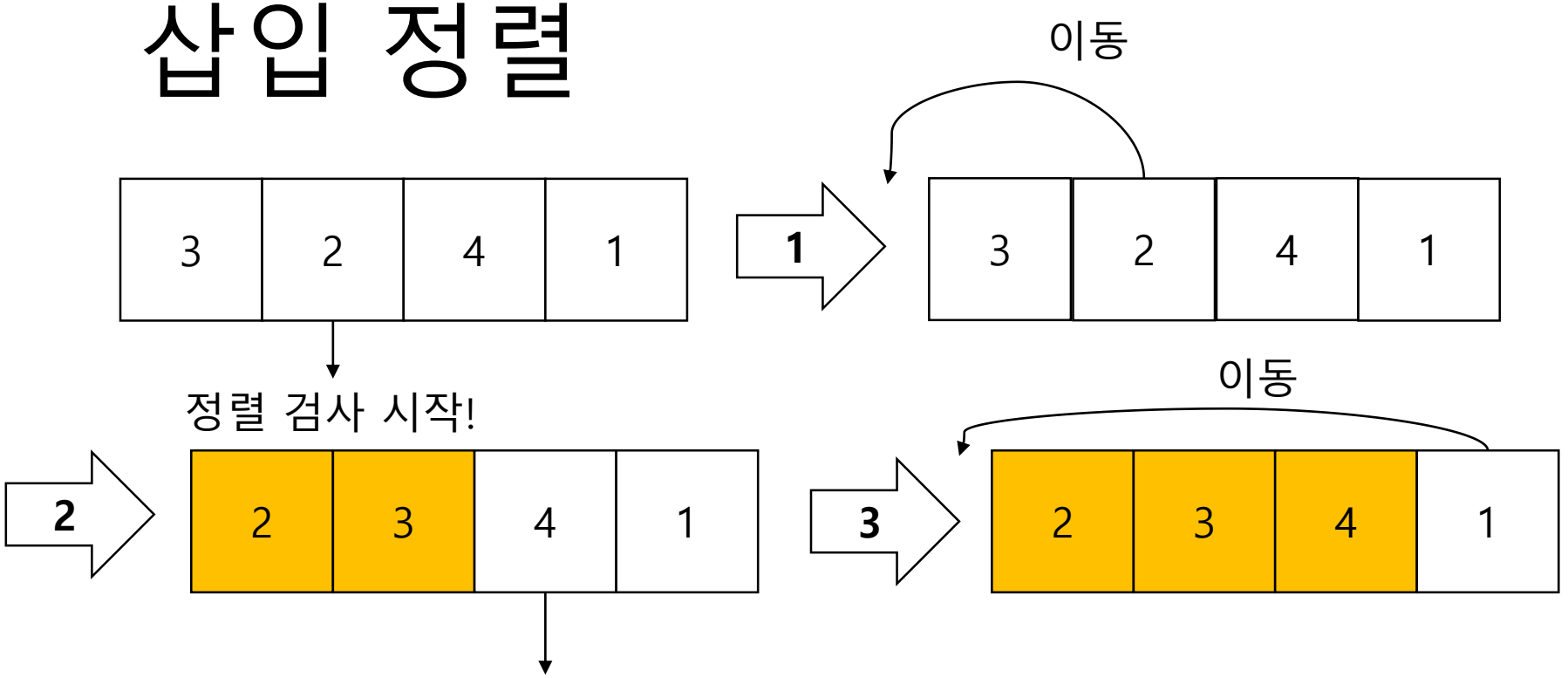
처음에 이런 상태로 데이터가 전달되어도 무조건 모든 과정을 거친다.
-> 너무 비효율적!

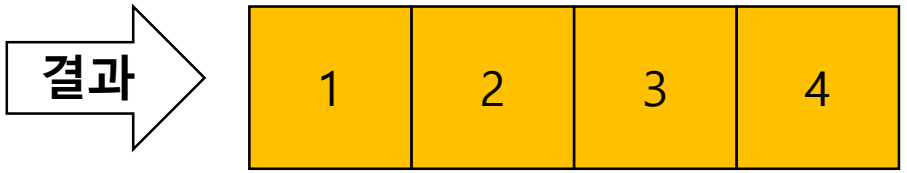
모든 검사를 다 했을 때 바꾼 데이터가 존재하지 않는다면 그것으로 정렬이 다 된 것이 아닐까?

# 선택 정렬



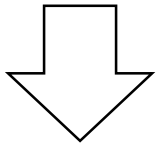
# 삽입 정렬



# 힙 정렬

최소 힙의 특징 : 부모 노드가 자식 노드보다 작거나 같아야 하며,
루트 노드의 우선순위가 가장 높다.

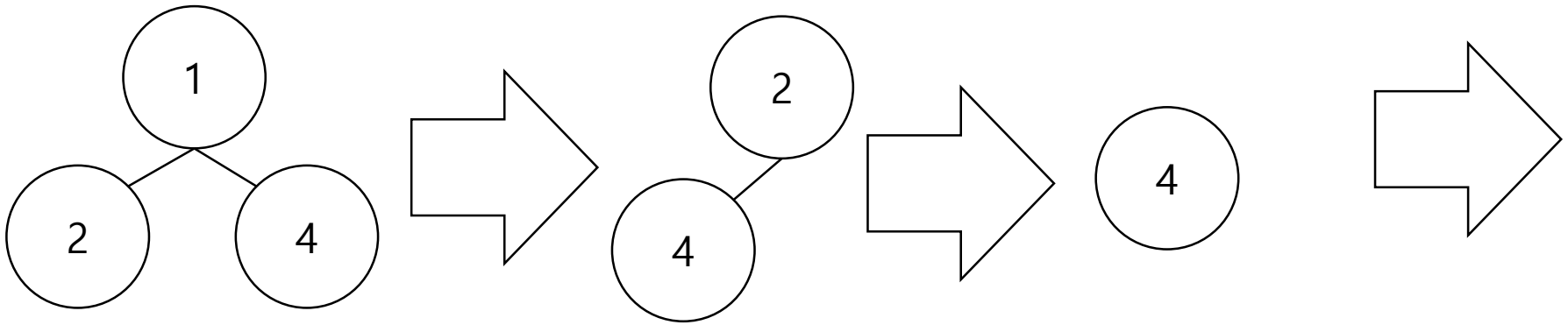


루트 노드에 있는 데이터를 순서대로 꺼내면 정렬이 된다.

출력 데이터 : 1

출력 데이터 : 1, 2

출력 데이터 : 1, 2, 4



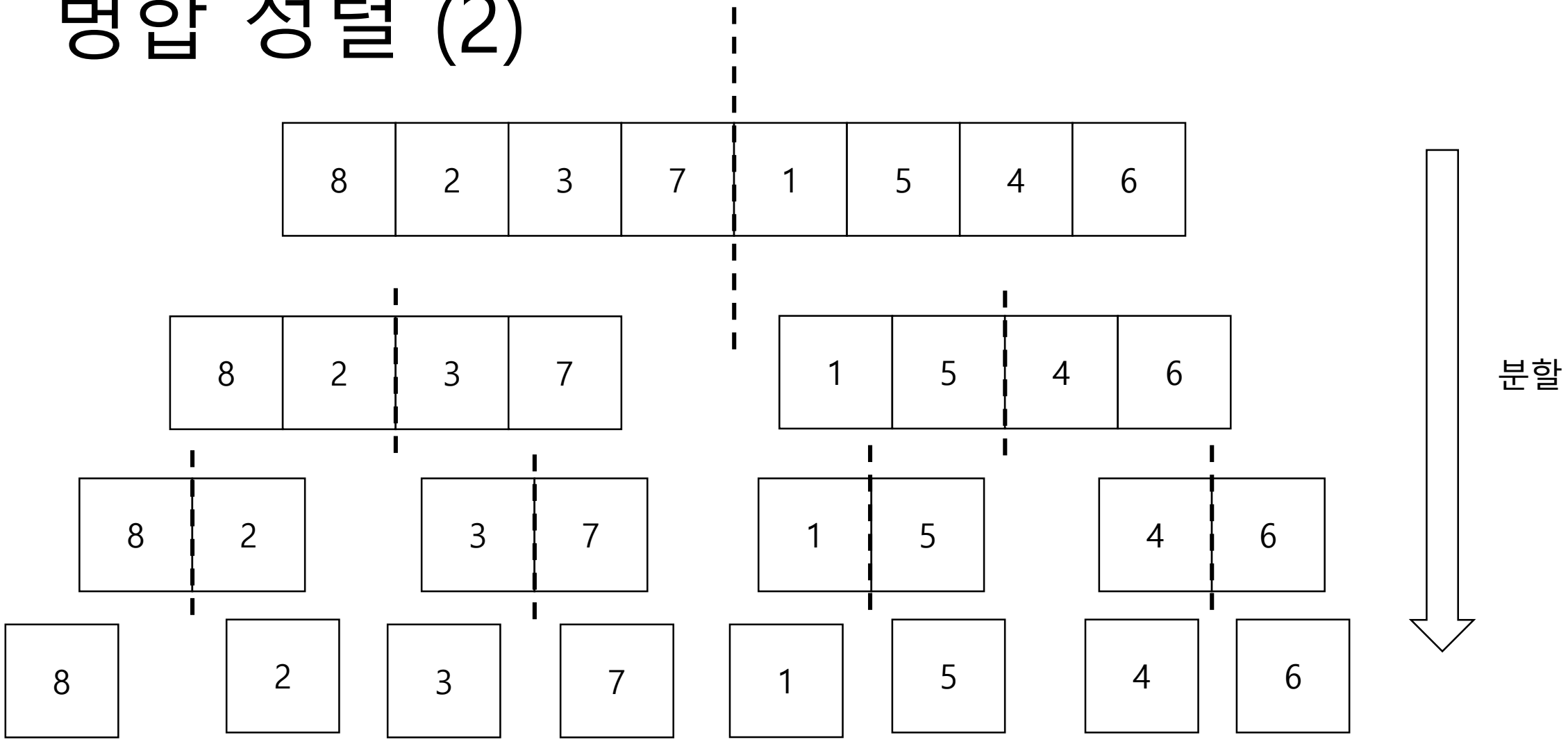
# 병합 정렬 (1)

병합 정렬은 총 3가지의 단계를 거쳐서 데이터들이 정렬된다.

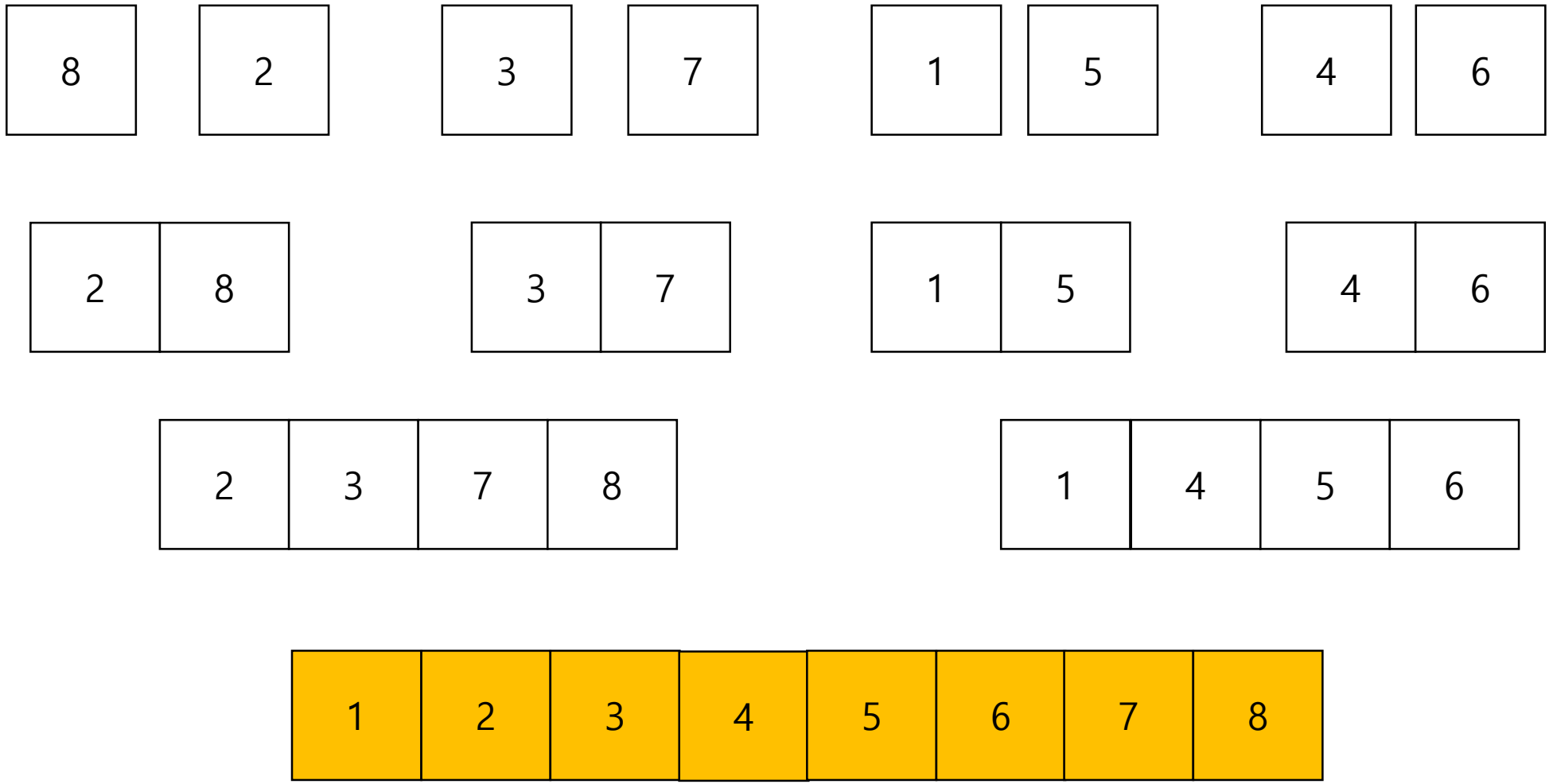
1단계 : 분할 - 문제를 계속해서 분할한다.
2단계 : 정복 – 분할된 문제를 해결한다.
3단계 : 결합 – 정복한 문제들을 하나로 결합한다.

말로만 하면 복잡해지기 쉬우니 그림으로 보면서 이해하도록 하자.

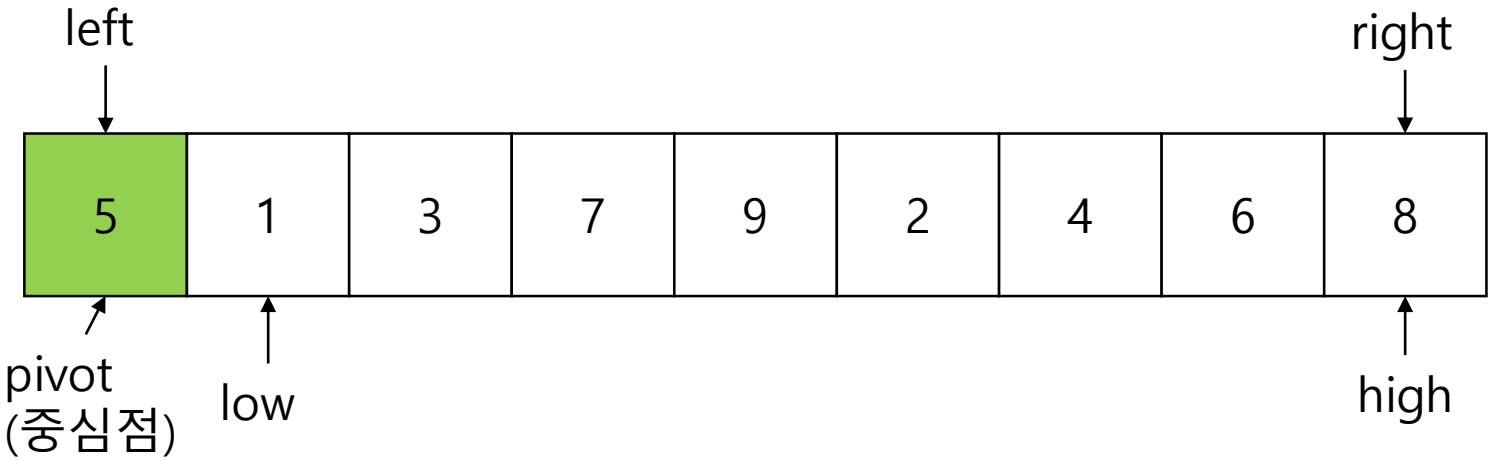
# 병합 정렬 (2)



# 병합 정렬 (3)



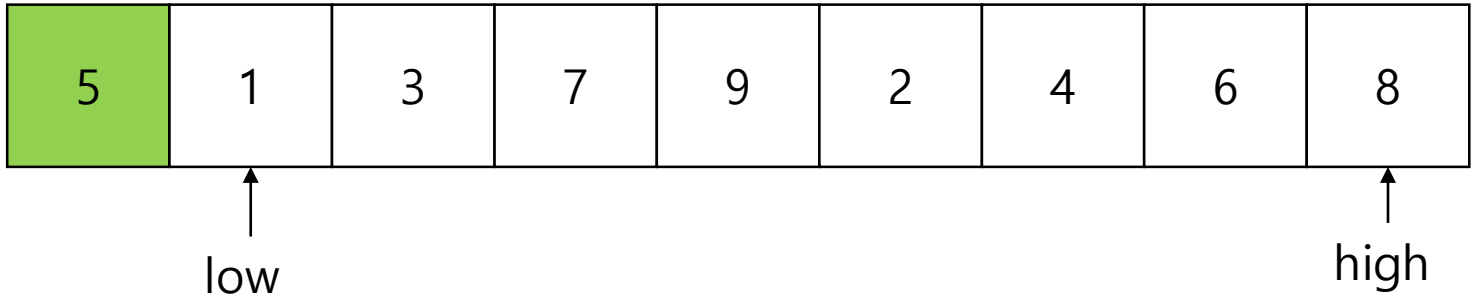
# 퀵 정렬 (1)




가장 왼쪽에 위치한 데이터를 퀵 정렬에 필요한 피벗으로 정한다고 가정하자.

low : 피벗을 제외한 가장 왼쪽에 위치한 지점을 가리키는 이름
high : 피벗을 제외한 가장 오른쪽에 위치한 지점을 가리키는 이름

# 퀵 정렬 (2)

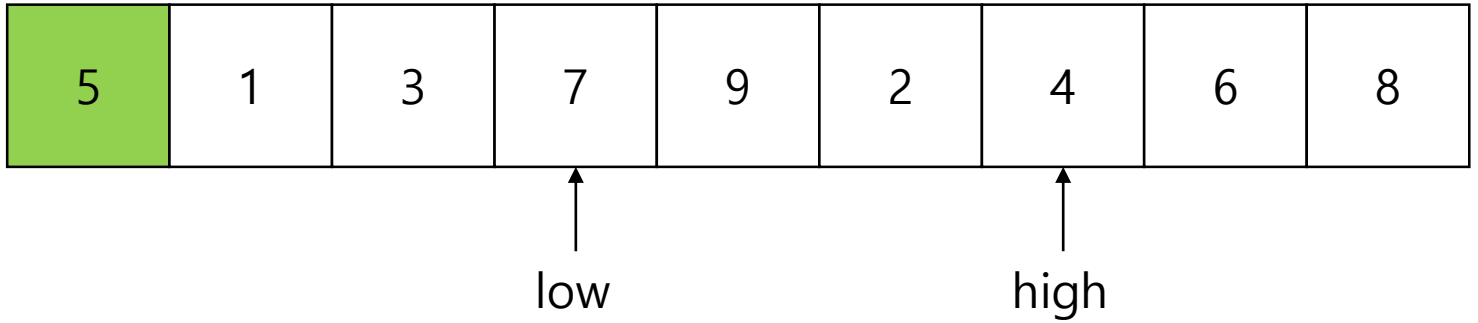



이 데이터들을 오름차순으로 정렬한다고 가정해보자.

low는 피벗보다 우선순위가 낮은 데이터를 만날 때까지 이동

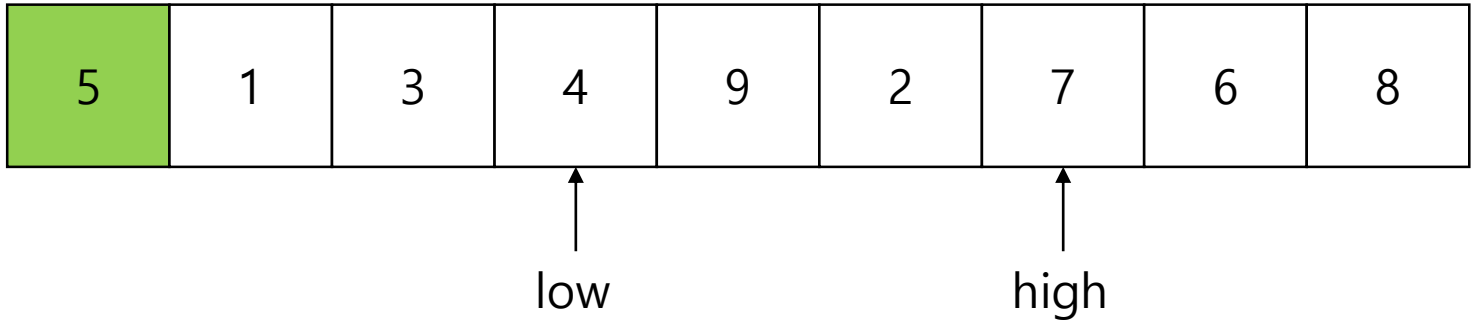
high는 피벗보다 우선순위가 높은 데이터를 만날 때까지 이동

# 퀵 정렬 (3)

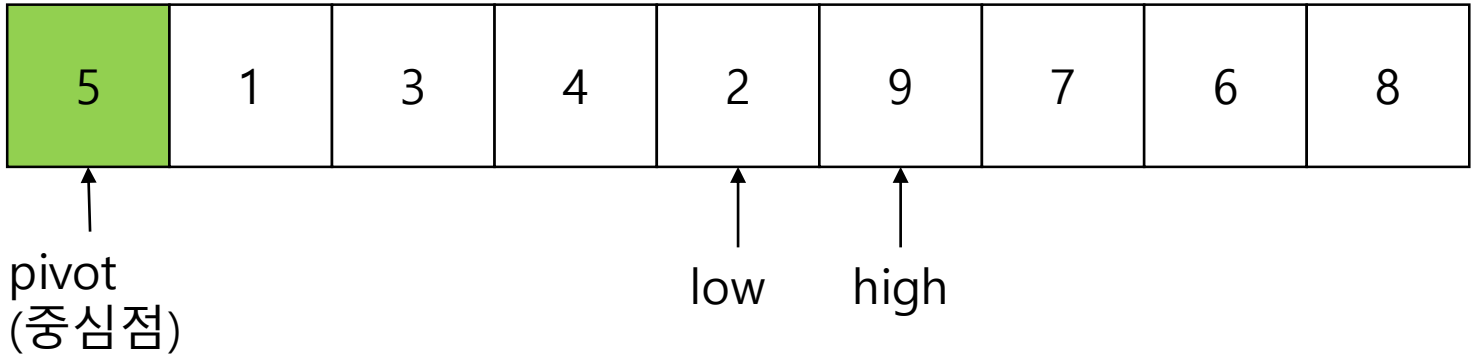



low와 high의 위치가 결정되었다면 그 둘의 위치를 변경한다.

# 퀵 정렬 (4)



# 퀵 정렬 (5)

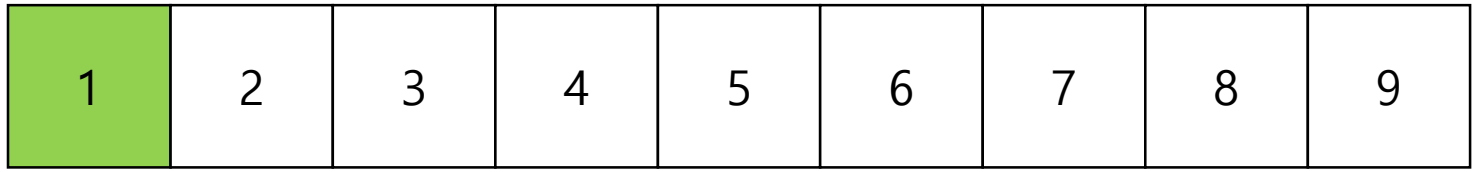



Low와 high를 규칙에 따라 움직이다 보면 둘이 교차하게 된다.
이 때는 high과 피벗을 교환한다.

이렇게 교환된 피벗은 정렬된 위치를 찾게 된다.

# 퀵 정렬 (6)

피벗은 주어진 데이터들의 중간 값일 경우, 정렬대상은 균등하게 나뉜다.

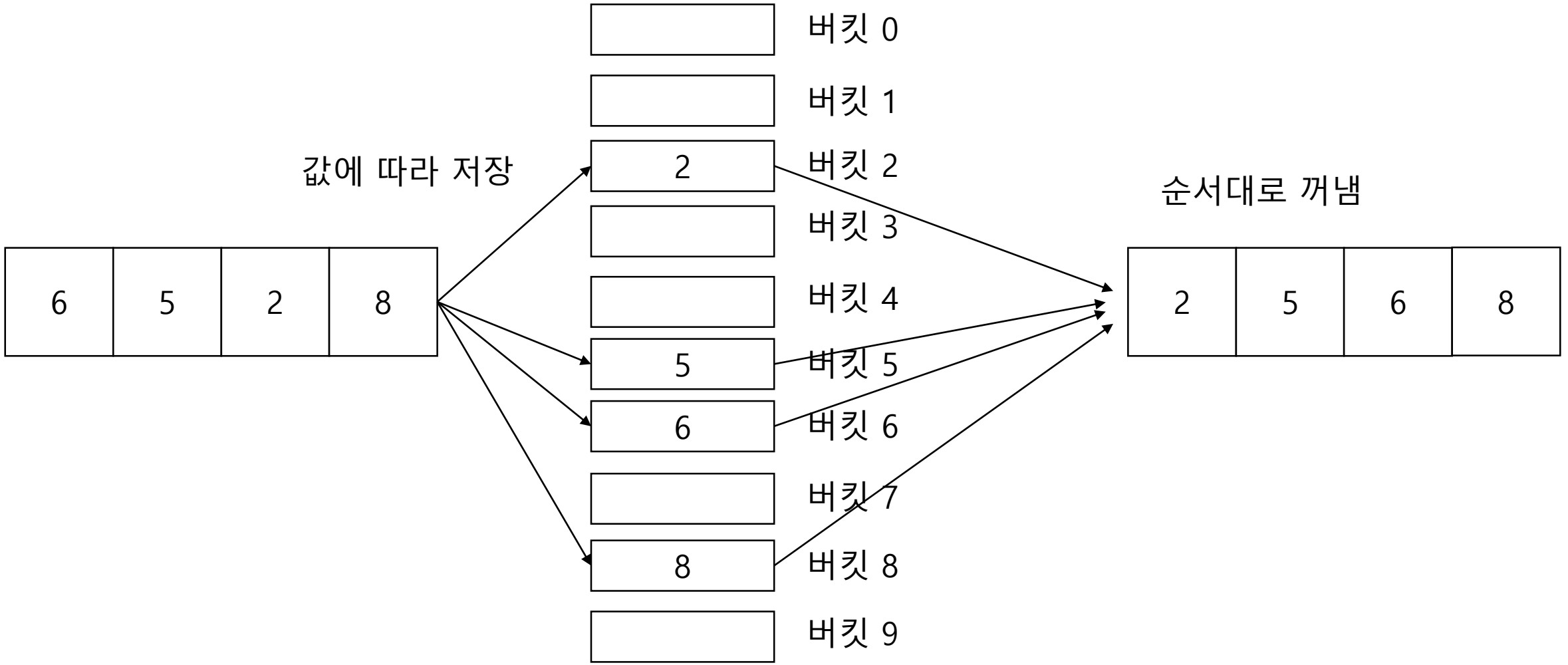



피벗을 가장 왼쪽의 데이터로 지정하는 경우
-> 1에서부터 시작해서 8까지 순서대로 피벗이 되어 정렬 과정을 거침

그래서 정렬대상에서 세 개의 데이터를 추출한다.
그리고 그 중에서 중간 값에 해당하는 것을 피벗으로 선택한다.

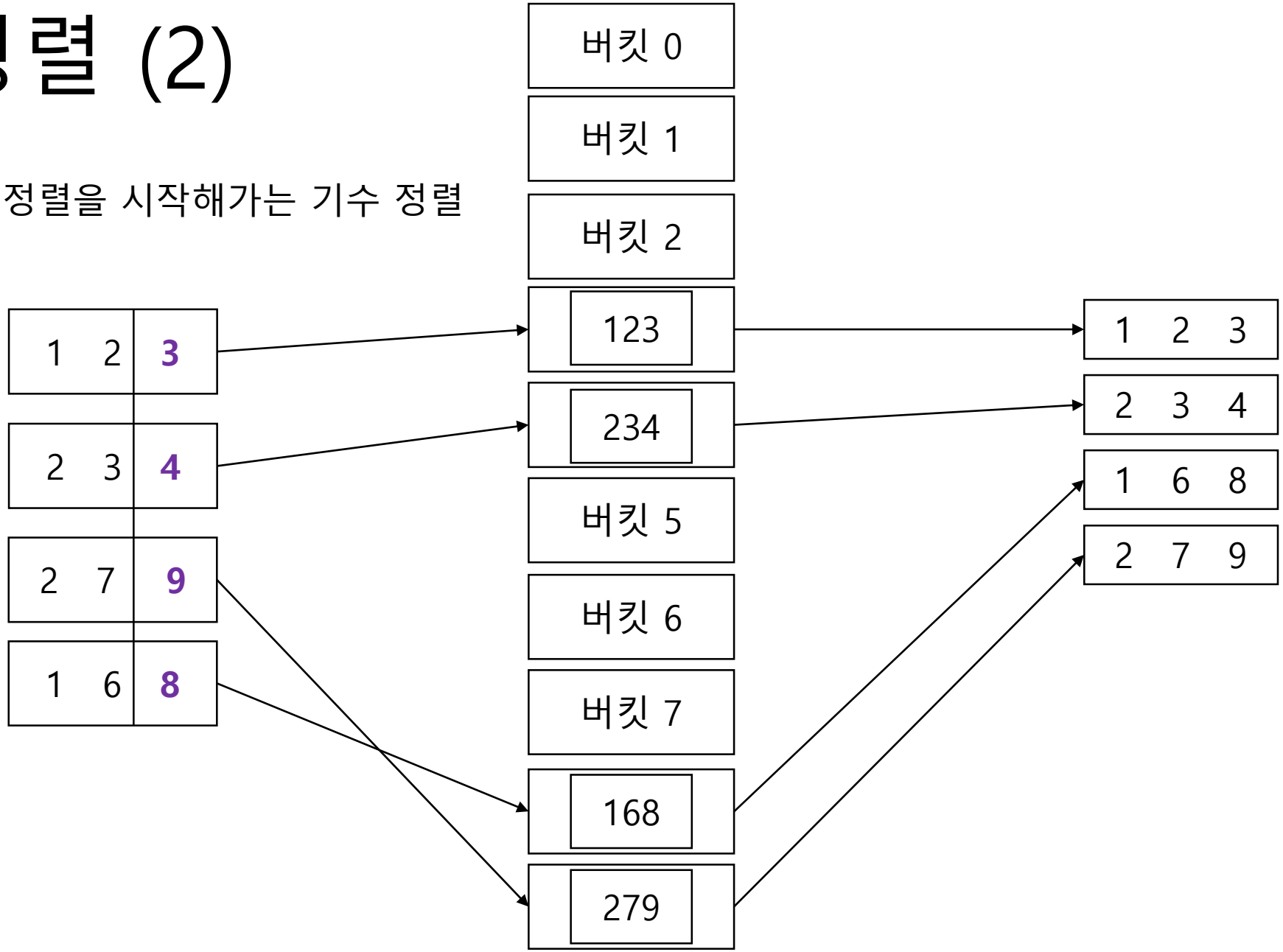
# 기수 정렬 (1)

보통 양의 정수들만 데이터 내에 존재할 때 사용하는 정렬 알고리즘이다.

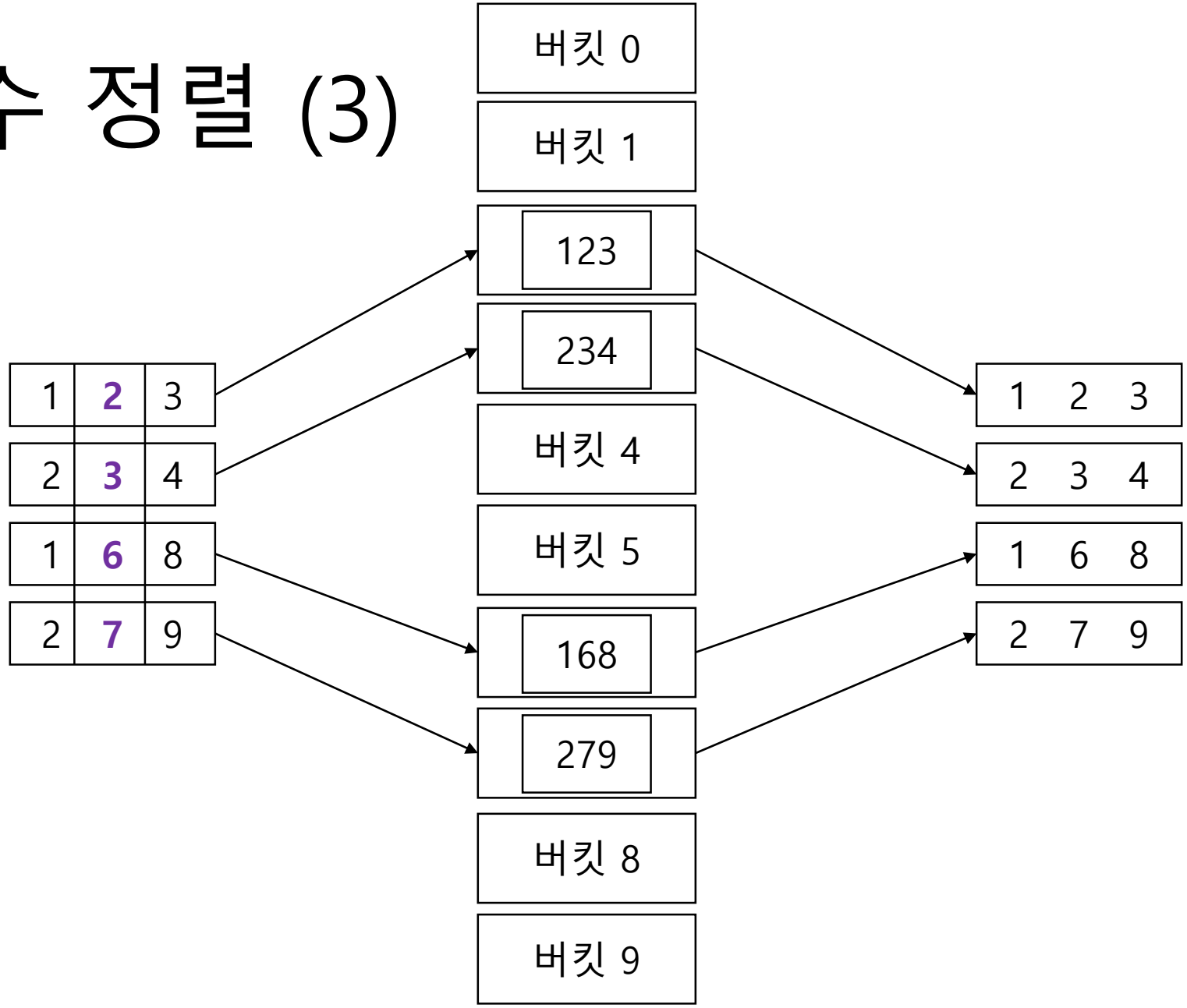


# 기수 정렬 (2)

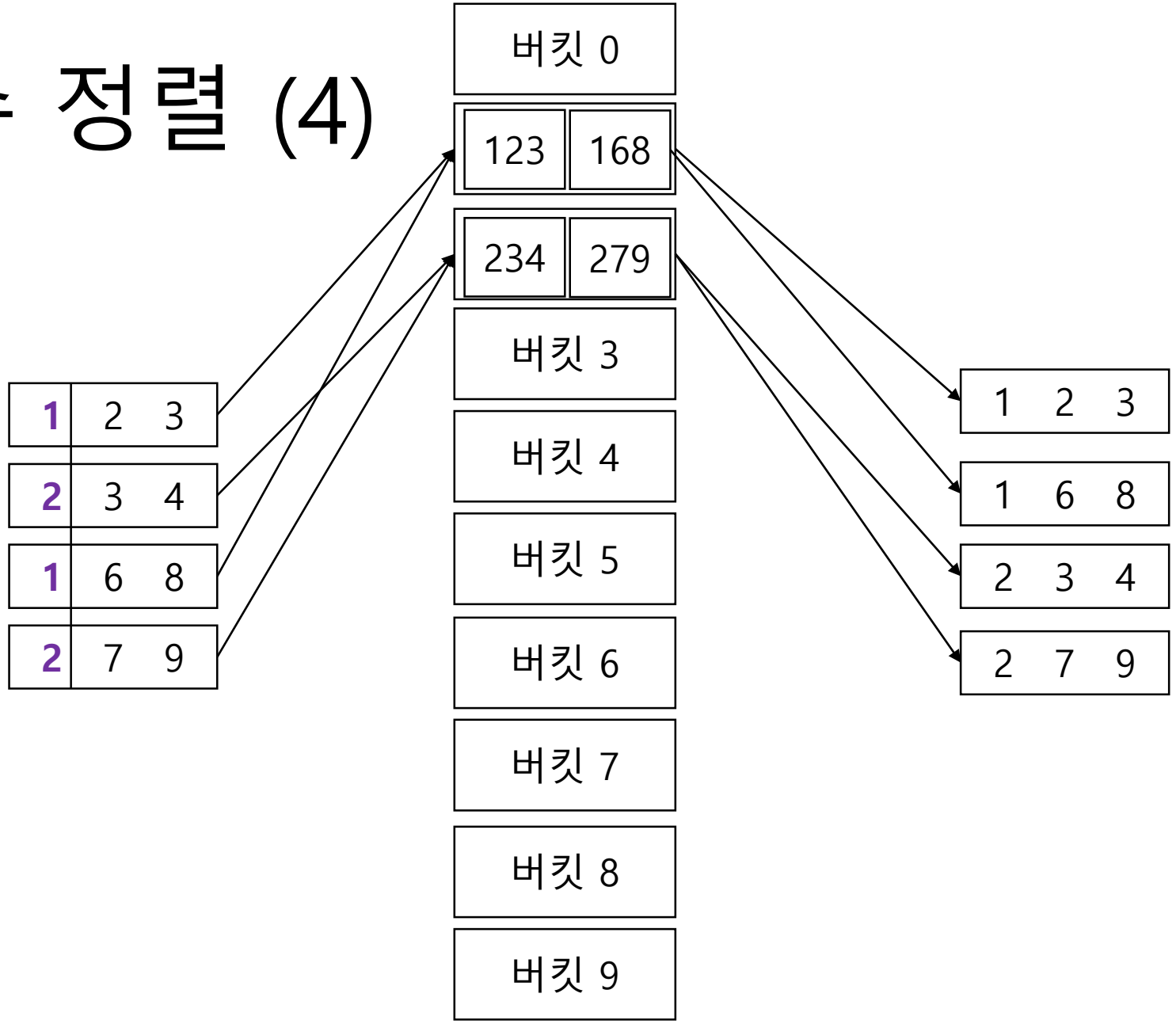
LSD 기수 정렬
일의 자리수부터 정렬을 시작해가는 기수 정렬



# 기수 정렬 (3)


버킷 0
버킷 1
123
234
버킷 4
버킷 5
168
279
버킷 8
버킷 9
1 2 3
2 3 4
1 6 8
2 7 9
1 2 3
2 3 4
1 6 8
2 7 9

# 기수 정렬 (4)


버킷 0
123
168
234
279
버킷 3
버킷 4
버킷 5
버킷 6
버킷 7
버킷 8
버킷 9
1 2 3
2 3 4
1 6 8
2 7 9
1 2 3
1 6 8
2 3 4
2 7 9

# 보간 탐색

보간 탐색은 중앙에서 탐색을 시작하지 말고, 탐색대상이 앞쪽에 위치해 있으면
앞쪽에서 탐색을 시작하는 방법

정수 12를 찾는다면……

10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

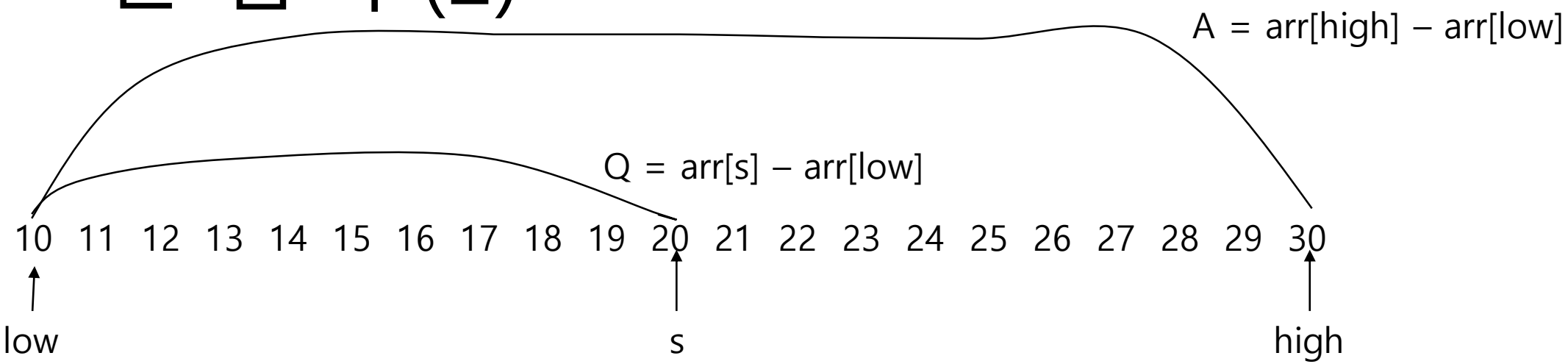
보간 탐색 탐색 위치

이진 탐색 탐색 위치

이진 탐색은 값에 상관없이 탐색위치를 결정한다.

보간 탐색은 그 값이 상대적으로 앞에 위치한다고 판단하면 앞쪽에서 탐색을 진행한다.

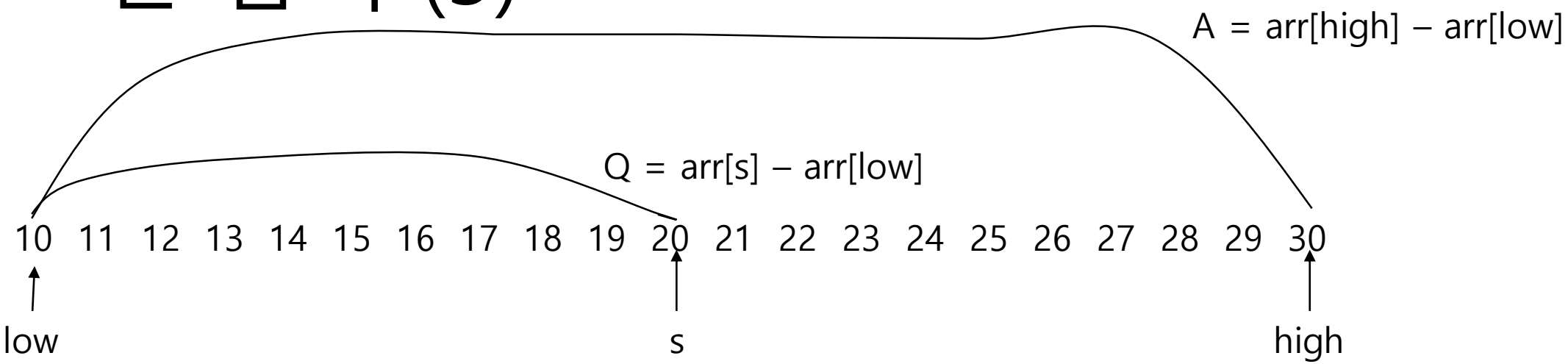
# 보간 탐색 (2)




low : 탐색대상의 처음
high : 탐색대상의 끝
s : 찾는 데이터가 저장된 위치의 인덱스 값

A : Q = (high - low) : (s – low)

# 보간 탐색 (3)




A : Q = (high - low) : (s – low)

s는 찾고자 하는 데이터의 인덱스 값 -> 위의 비례식을 s에 대한 식으로 정리

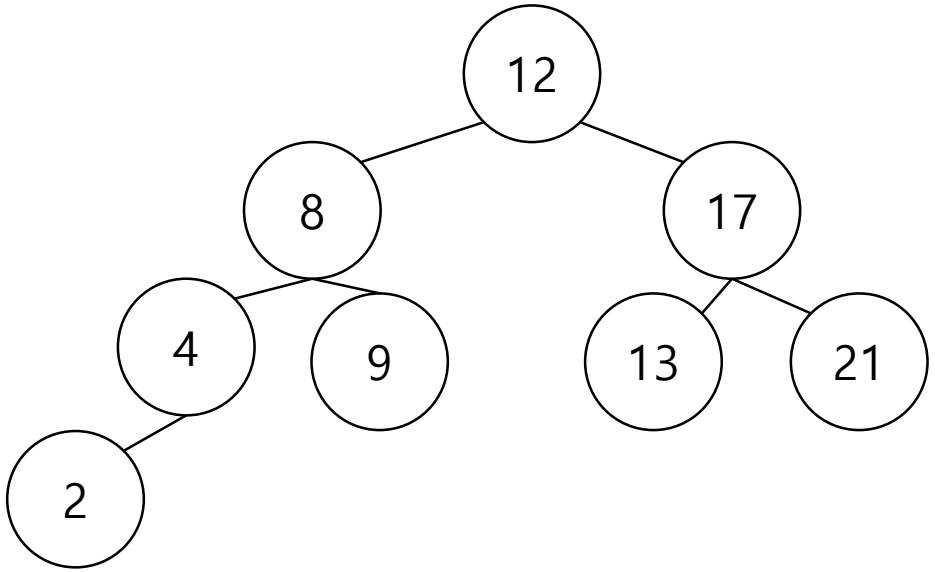
s = $\frac{Q}{A}(high - low) + low$

그리고 찾는 값 arr[s]를 x라 하였을 때 최정적으로 다음과 같이 정리된다.

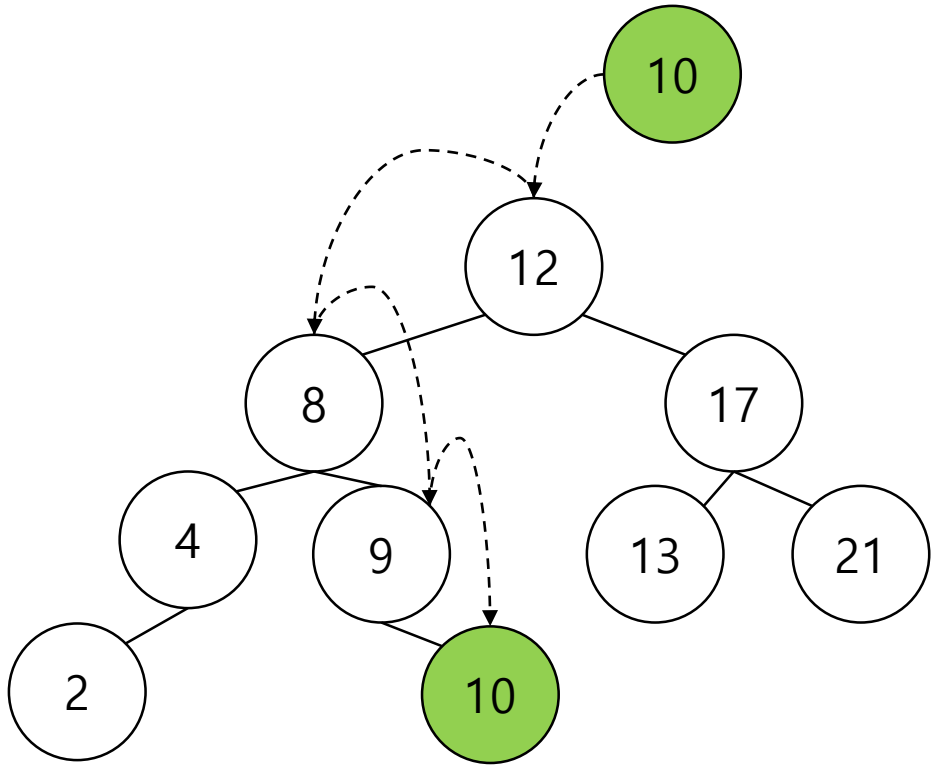
s = $\frac{x - arr[low]}{arr[high] - arr[low]}(high - low) + low$

# 이진 탐색 트리 (BST)

- 이진 탐색 트리의 노드에 저장된 키(key)값은 유일하다. -> 키 값은 중복될 수 없다.
- 루트 노드의 키가 왼쪽 서브 트리를 구성하는 어떠한 노드의 키보다 크다.
- 루트 노드의 키가 오른쪽 서브 트리를 구성하는 어떠한 노드의 키보다 작다.
- 왼쪽과 오른쪽 서브 트리도 이진 탐색 트리이다.



# 이진 탐색 트리 삽입





그렇다면 20은 어디에
삽입되어야 할까?

# 이진 탐색 트리의 삭제 (1)

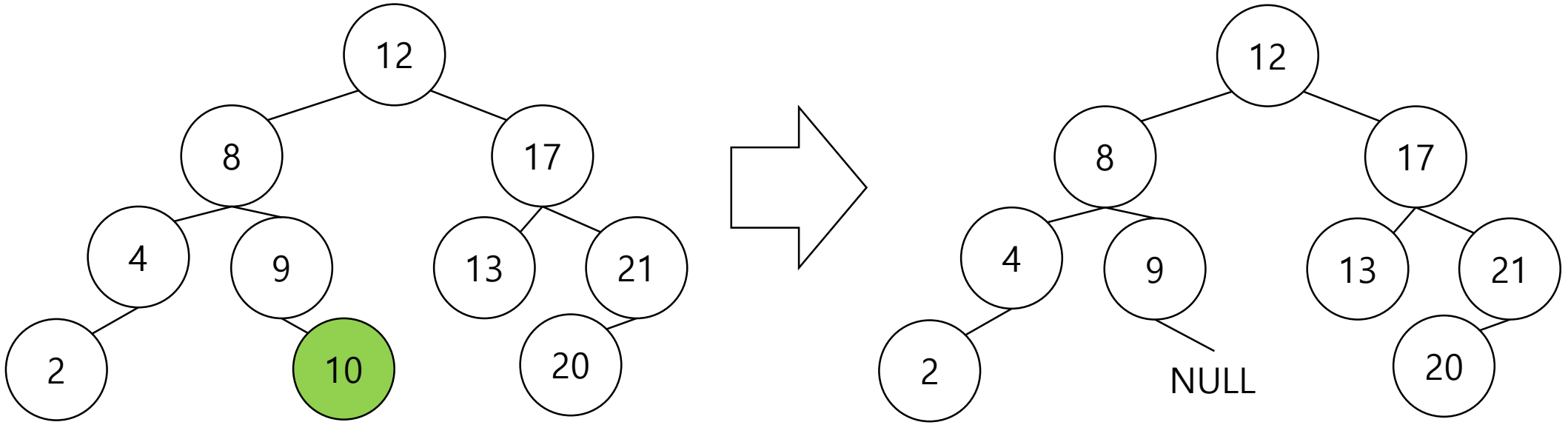
삭제 시 남은 트리는 모두 이진 탐색 트리의 규칙을 준수하고 있는 상태여야 한다.

삭제를 하는 과정에서 다음과 같은 세 가지 경우가 발생할 수 있다.

Case 1 : 단말 노드를 삭제하는 경우
Case 2 : 자식 노드가 1개만 있는 노드를 삭제하는 경우
Case 3 : 자식 노드가 2개 다 존재하는 노드를 삭제하는 경우

# 이진 탐색 트리의 삭제 (2)

단말 노드를 삭제하는 경우

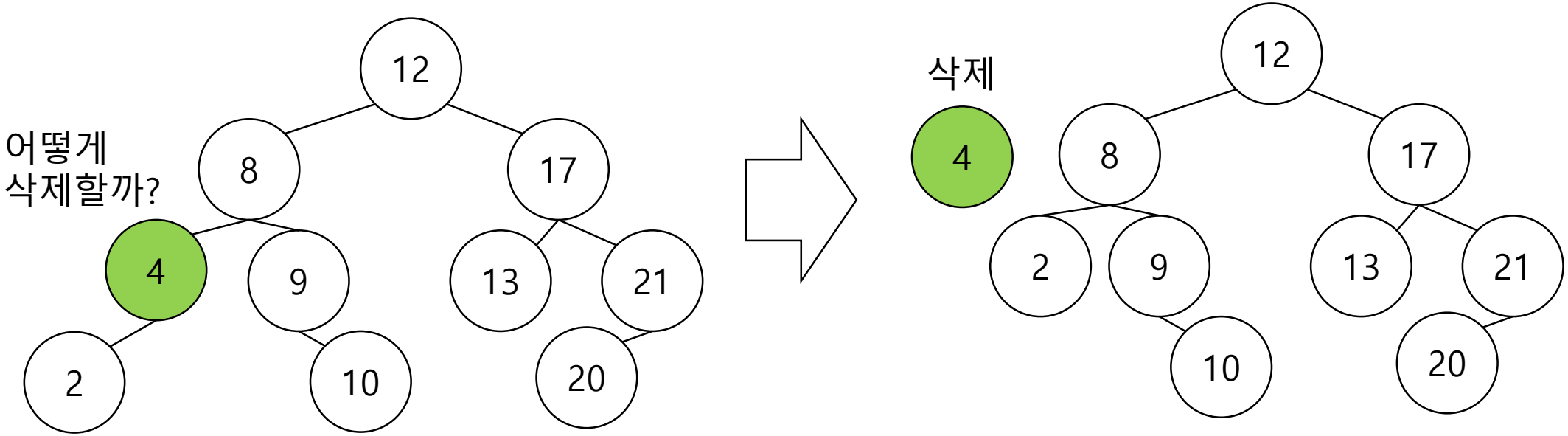



어떻게 삭제할까?

해당 노드를 삭제하고 NULL로 연결해주면 된다.

# 이진 탐색 트리의 삭제 (3)

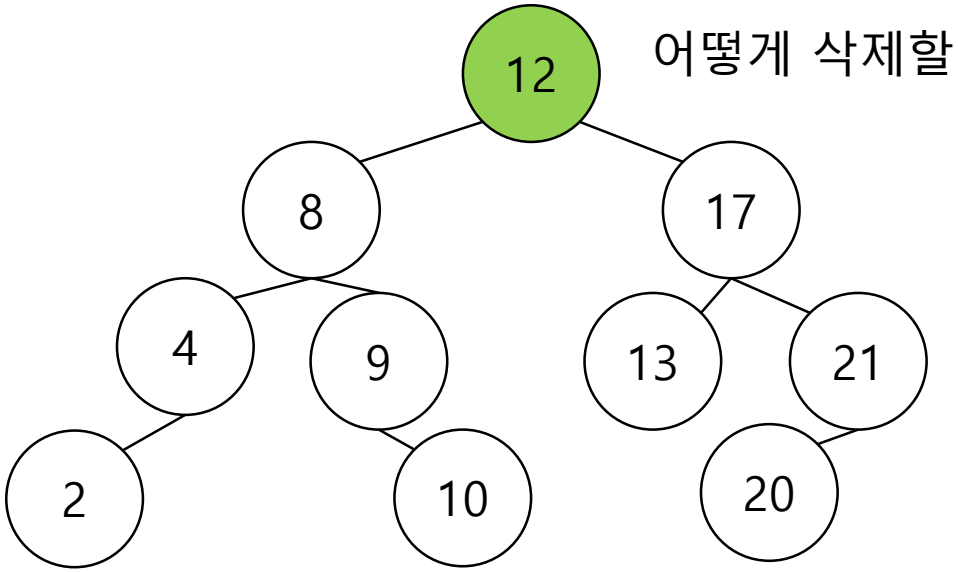
자식 노드가 1개 있는 노드를 삭제하는 경우




삭제하고자 하는 노드의 자식을 삭제하고자 하는 노드의 부모와 연결해준 다음 삭제한다.

# 이진 탐색 트리의 삭제 (4)

자식 노드가 2개 존재하는 노드를 삭제하는 경우



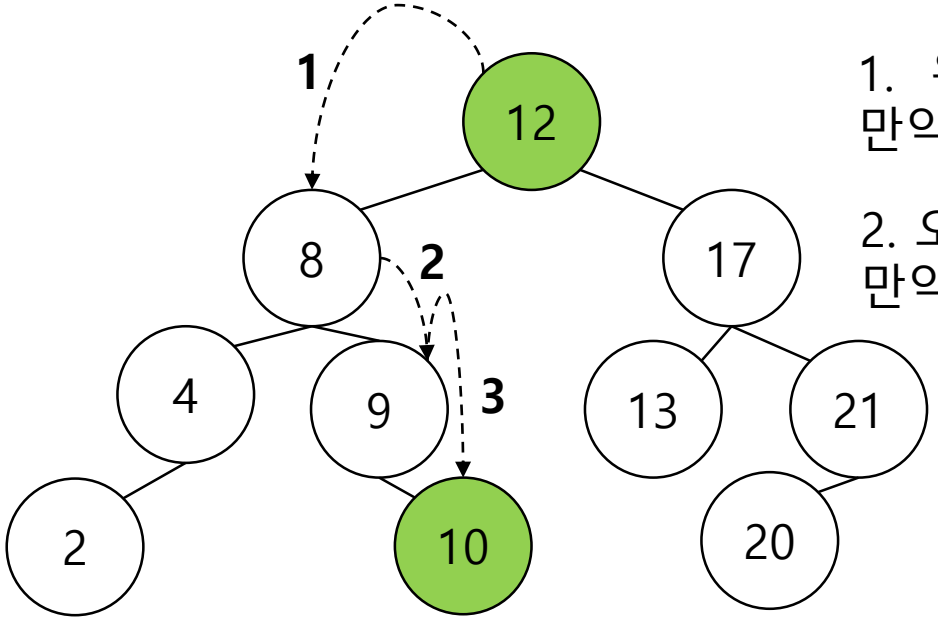

어떻게 삭제할까?

우선 자신의 자리를 대체할 후계자를 선정해야 한다.
그런데 후계자를 찾는 방법은 크게 2가지가 있다.

1. 자신의 왼쪽 서브트리에서 후계자를  찾는 경우
2. 자신의 오른쪽 서브트리에서 후계자를 찾는 경우

# 이진 탐색 트리의 삭제 (5)

왼쪽 서브트리에서 후계자를 찾는 경우

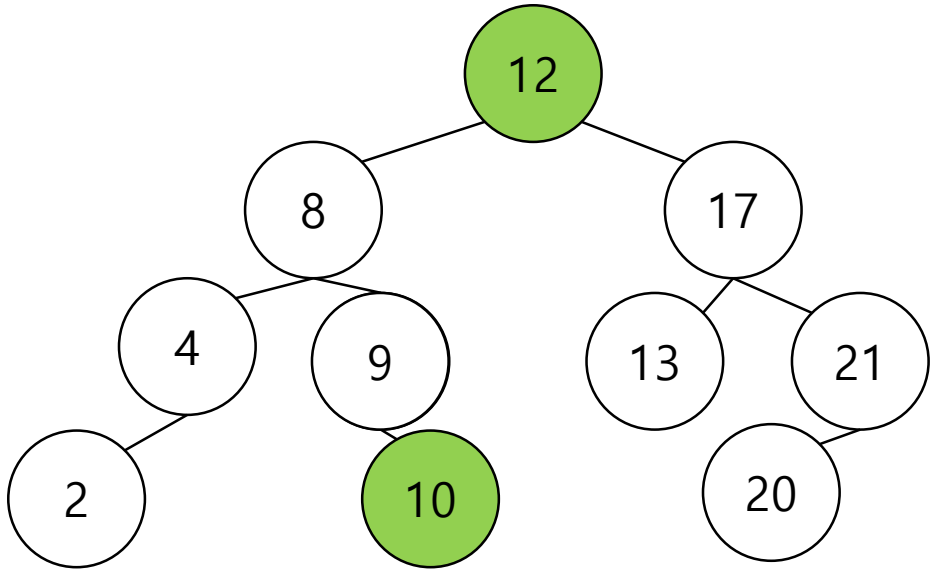



1. 왼쪽 서브트리로 이동한다.
만약 해당 노드의 오른쪽 자식이 없다면 해당 노드가 후계자가 된다.

2. 오른쪽 서브트리로 이동한다.
만약 해당 노드의 오른쪽 자식이 없다면 해당 노드가 후계자가 된다.

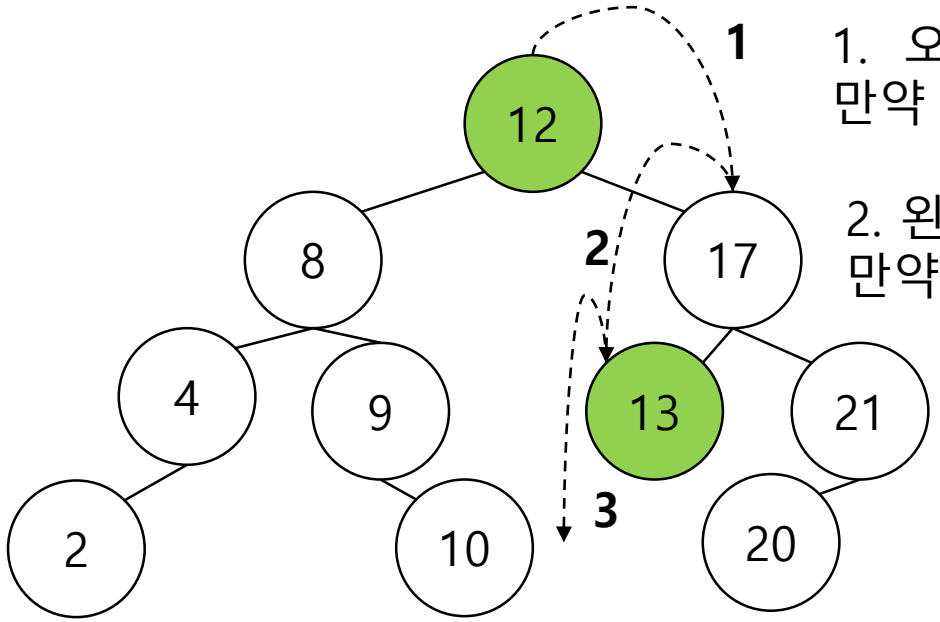
3. 2번의 과정을 계속해서 반복한다.

# 이진 탐색 트리의 삭제 (6)




1. 후계자의 값을 삭제할 노드의 값에 대입한다.
2. 후계자의 위치에 후계자의 자식을 둔다.

# 이진 탐색 트리의 삭제 (7)

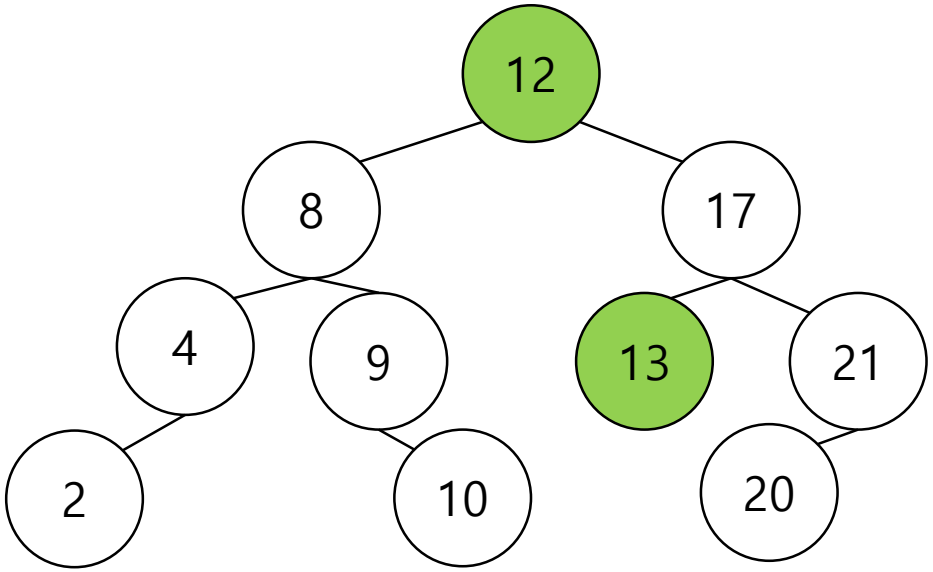



1. 오른쪽 서브트리로 이동한다.
만약 해당 노드의 왼쪽 노드가 없다면 해당 노드가 후계자가 된다.

2. 왼쪽 서브트리로 이동한다.
만약 해당 노드의 왼쪽 노드가 없다면 해당 노드가 후계자가 된다.

3. 2번의 과정을 반복한다.

# 이진 탐색 트리의 삭제 (8)

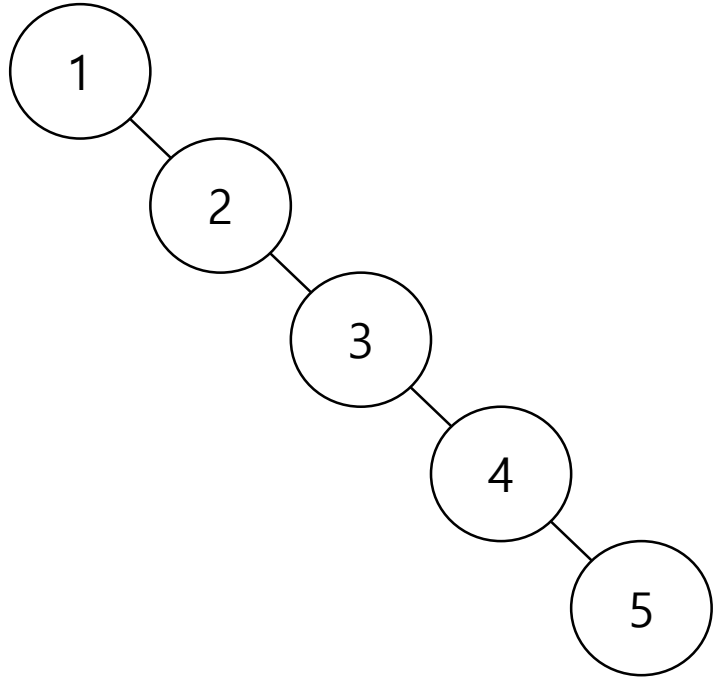



1. 후계자의 값을 삭제할 노드의 값에 대입한다.
2. 후계자의 위치에 후계자의 자식을 둔다.

# AVL 트리 (1)

이진 탐색 트리는 다음과 같은 단점을 가지고 있다.

다음과 같은 모양으로 이진 탐색 트리가 형성되면 효율성이 떨어진다.



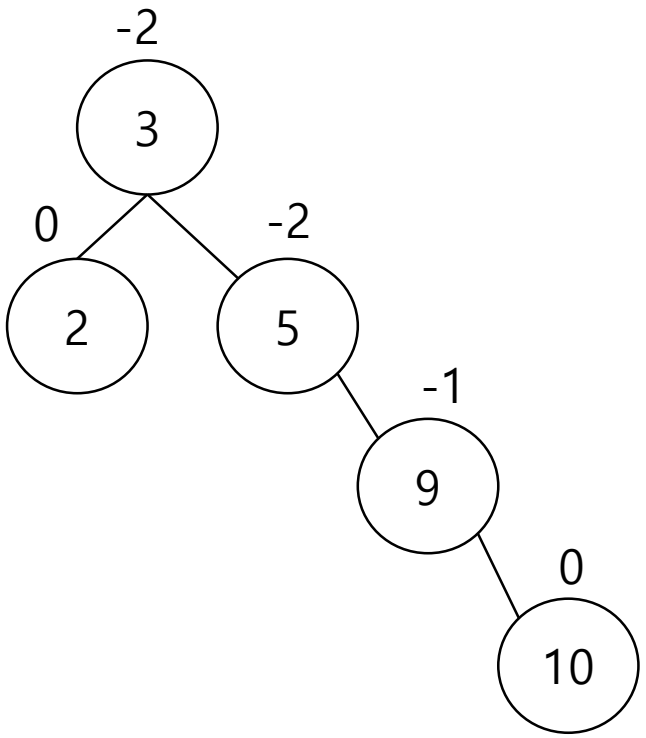

다음과 같은 이진 탐색 트리의 불균형을 맞추기 위한 기법은 여러가지가 존재한다.

- AVL 트리
- 2-3 트리
- 2-3-4 트리
- Red-Black 트리
- B 트리

# AVL 트리 (2)

노드가 추가 및 삭제될 때, 균형상태를 파악해서 스스로 그 구조를 변형해 균형을 잡는다.

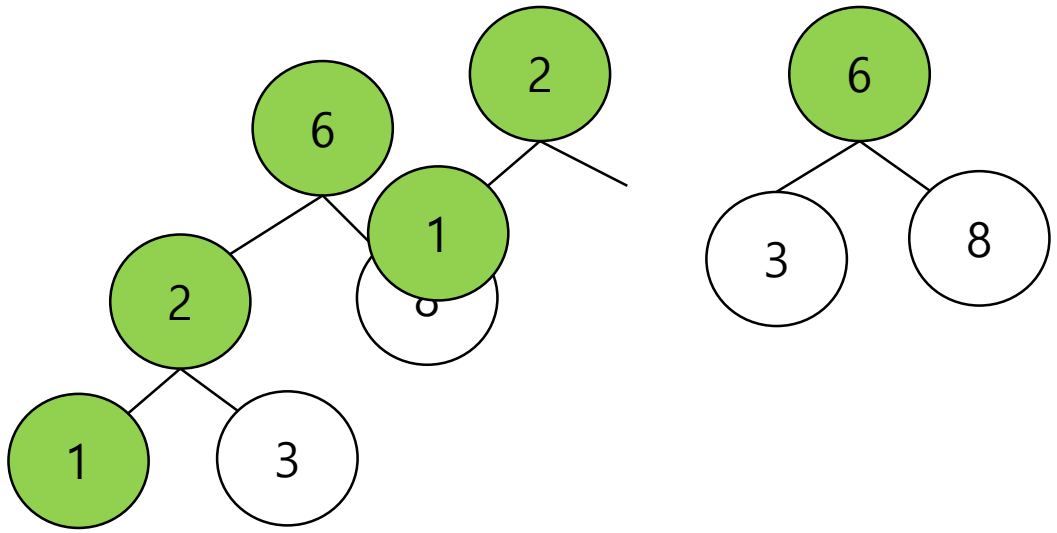
균형의 정도를 표현하기 위해 균형인수를 사용한다.
**균형인수 = 왼쪽 서브 트리의 높이 – 오른쪽 서브 트리의 높이**




균형을 잡는 과정을 '리벨런싱 ' 이라 가정하고
리벨런싱 과정을 알아보자!

# 언벨런싱 유형 1 – LL 유형

부모노드로부터 자식이 left에 있고, 그 노드의 자식도 left에 있는 경우



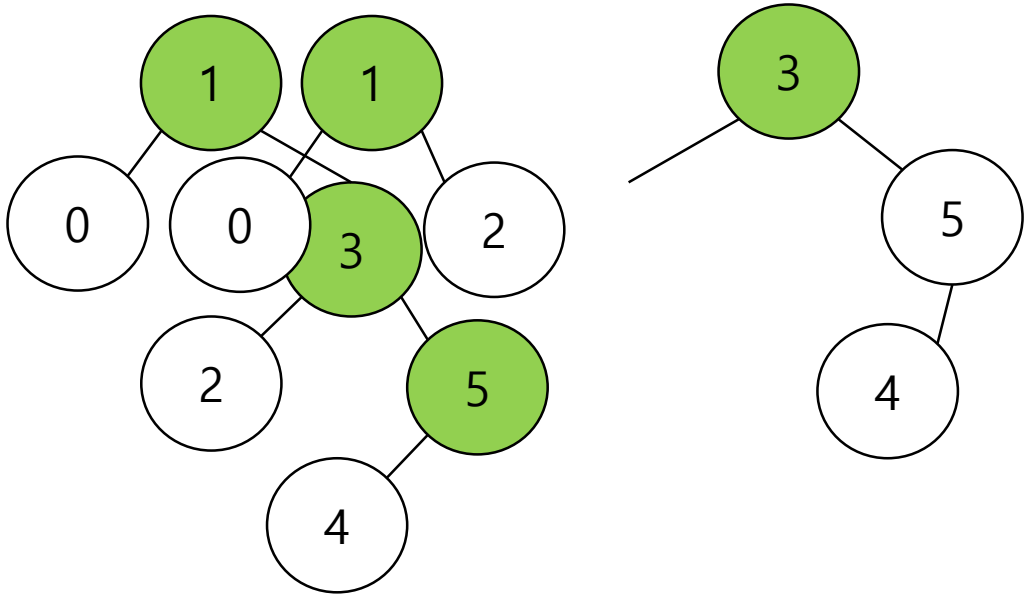

해결방법

6을 pNode, 5를 cNode로 가정하자.

1. cNode의 오른쪽 자식을 pNode의 왼쪽에 연결한다.
2. cNode의 오른쪽에 pNode를 연결한다.

이러한 방법을 LL 회전이라 한다.

# 언벨런싱 유형 – RR 유형







부모 노드로부터의 자식이 right에 있고, 그 노드의 자식도 right에 달려있는 경우

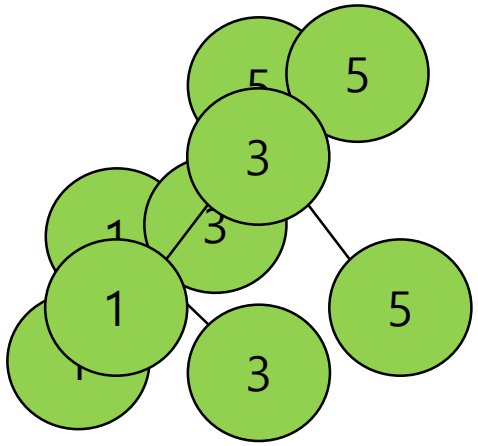
해결방법

1을 pNode로, 3을 cNode로 가정하자.

1. pNode의 오른쪽을 cNode의 왼쪽 자식과 연결한다.
2. cNode의 왼쪽에 pNode를 연결한다.

이러한 방법을 RR 회전이라 한다.

# 언벨런싱 유형 – LR 유형



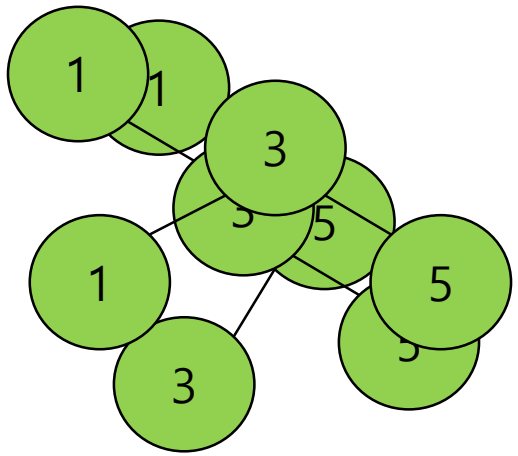

최상위 노드의 자식은 left에, 그 노드의 자식은 right에 달려있는 유형

해결방법

1. 노드 1을 최상위 노드로 해서 RR 회전을 한다.
2. RR 회전된 결과로 인해 LL 유형이 형성된다.
3. 형성된 LL 유형은 LL 회전으로 해결한다.

이러한 방법을 LR 회전이라 한다.

# 언벨런싱 유형 - RL 유형




최상위 노드의 자식이 right에, 그 노드의 자식은 left에 존재하는 유형

해결방법

1. 5를 최상위 노드로 해서 LL 회전을 한다.

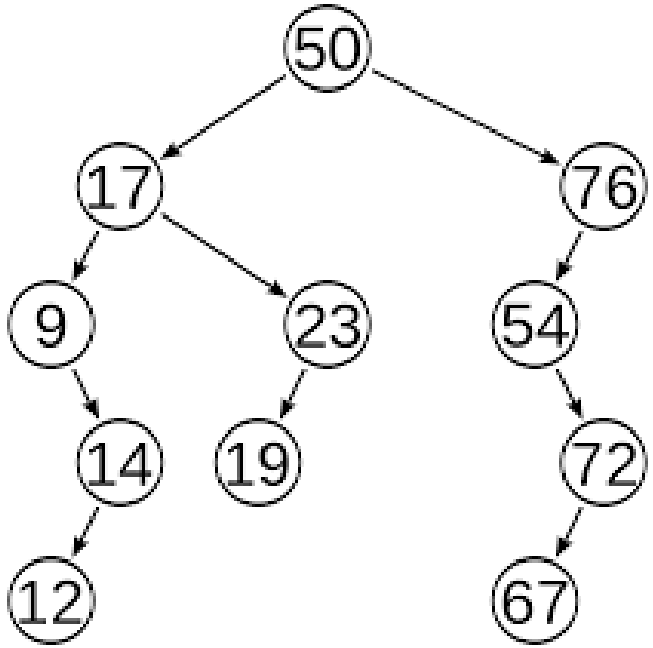
2. 그렇게 하면 RR 유형이 형성된다.

3. 형성된 RR 유형을 RR 회전을 통해 해결한다.

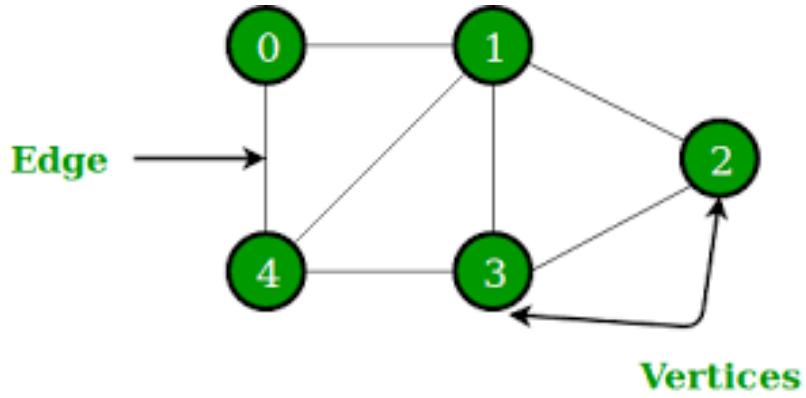
이러한 방법을 RL회전이라 한다.

# 언벨런싱에서 주의할 점

언벨런싱은 언제나 최하위 노드에서부터 루트 노드로 균형인수를 검사한다.
  -> 후위 순회 기법



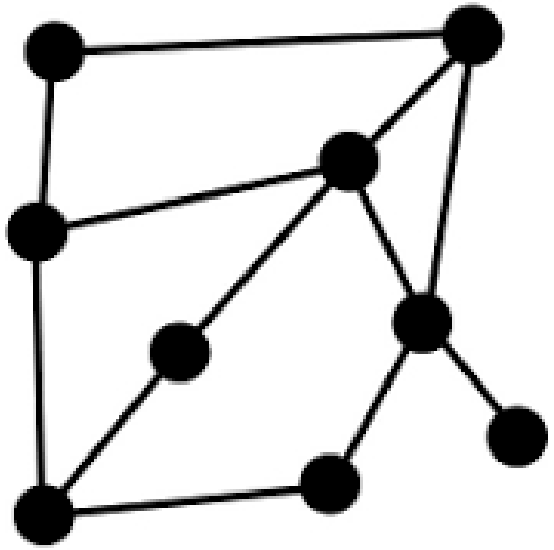
# 그래프




그래프에서 점(목적지)를 의미하는 것을 **정점**이라 한다.

그래프에서 정점을 연결한 선을 **간선**이라 한다.

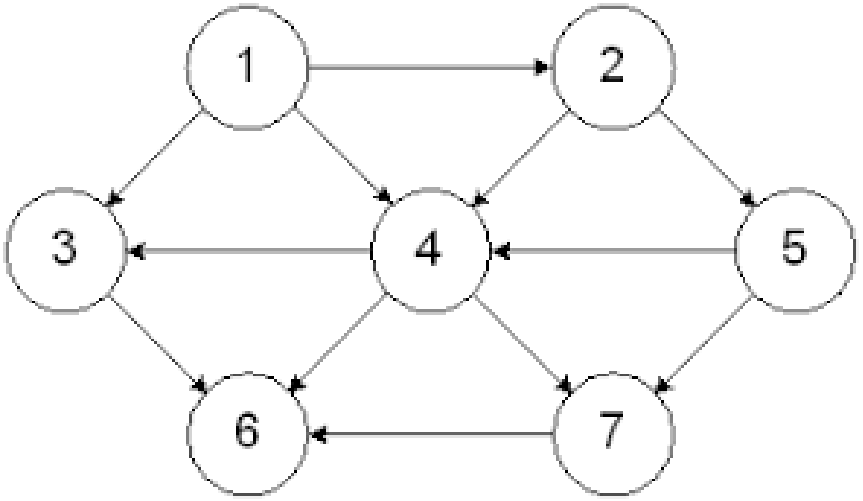
# 무방향 그래프



간선의 방향이 정해지지 않은(없는) 그래프

한 간선을 통해 연결되어 있는 두 정점을 제약없이 지나다닐 수 있다.

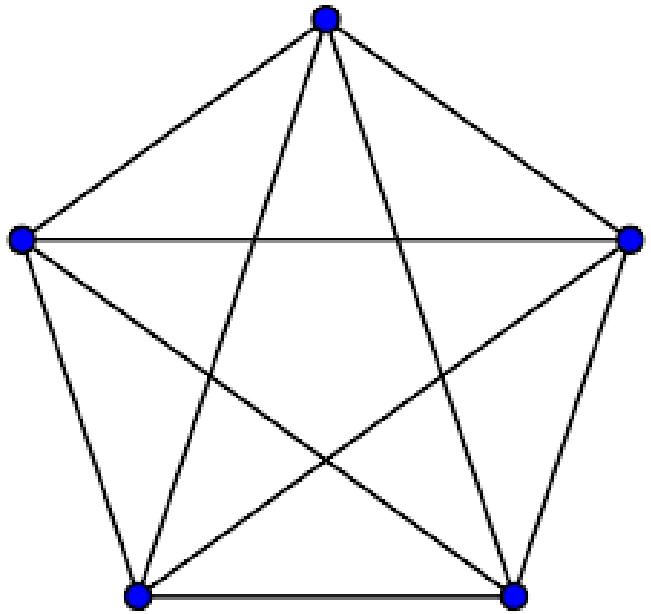
# 방향 그래프




연결관계에 있어 방향이 정해져 있는 그래프

간선의 방향이 정해져 있어 일방통행이다.

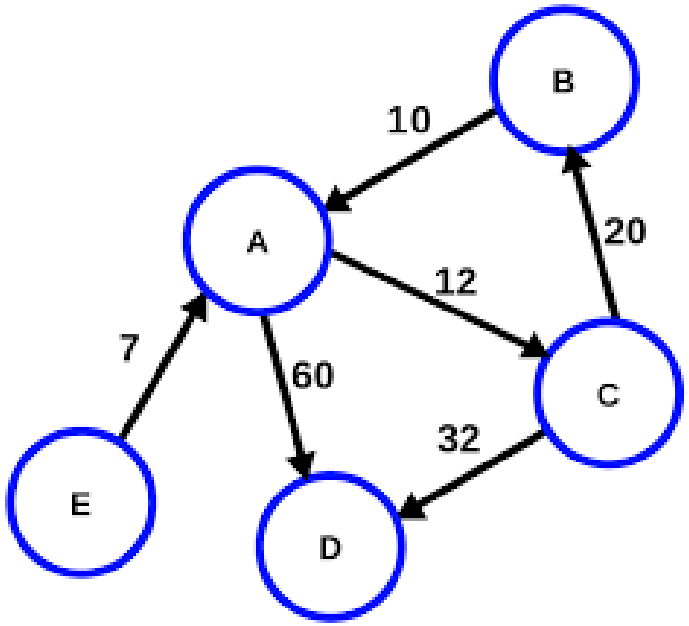
# 완전 그래프



무방향 그래프, 방향 그래프는 상관없이
다음 조건을 만족하면 완전 그래프라고 한다.

각각의 정점에서 모든 정점을 연결한 그래프일 것!

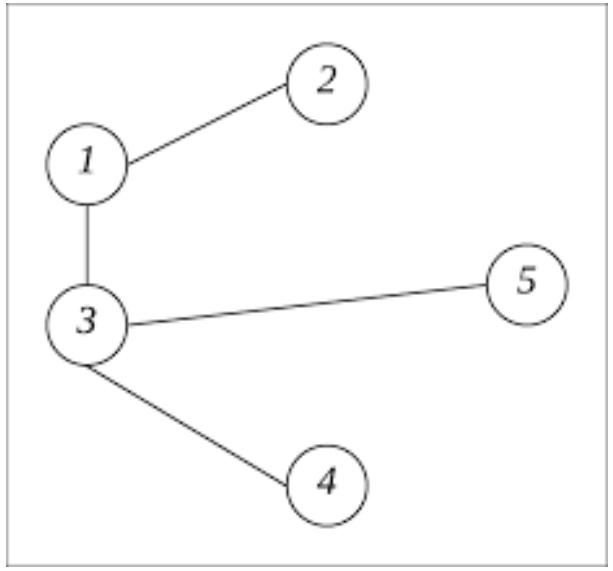
# 가중치 그래프




간선에 가중치를 두어서 구성한 그래프

주로 수학 관련 문제에서 많이 등장한다.

# 무방향 그래프의 표현

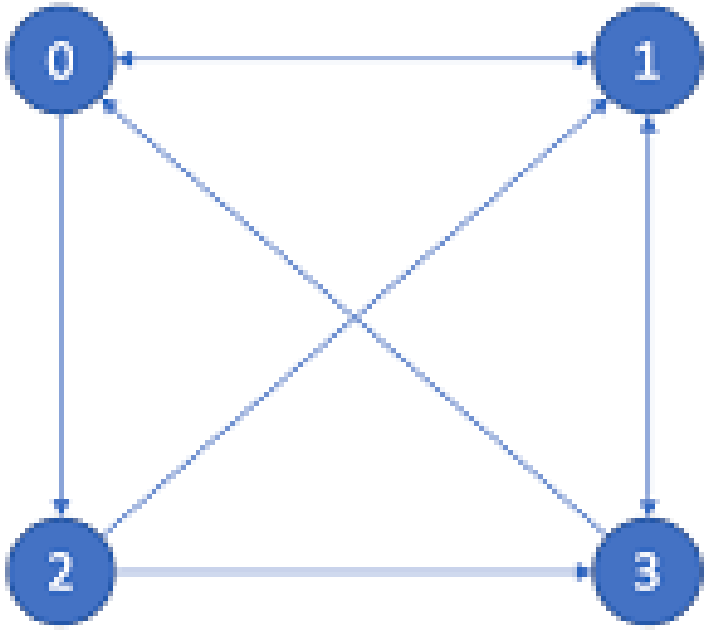



| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | ∞ | ∞ |
| 2 | 1 | 0 | ∞ | ∞ | ∞ |
| 3 | 1 | ∞ | 0 | 1 | 1 |
| 4 | ∞ | ∞ | 1 | 0 | ∞ |
| 5 | ∞ | ∞ | 1 | ∞ | 0 |

# 인접 행렬 표현의 그래프화

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | ∞ | ∞ |
| 2 | 1 | 0 | ∞ | ∞ | ∞ |
| 3 | 1 | ∞ | 0 | 1 | 1 |
| 4 | ∞ | ∞ | 1 | 0 | ∞ |
| 5 | ∞ | ∞ | 1 | ∞ | 0 |

# 방향 그래프의 표현

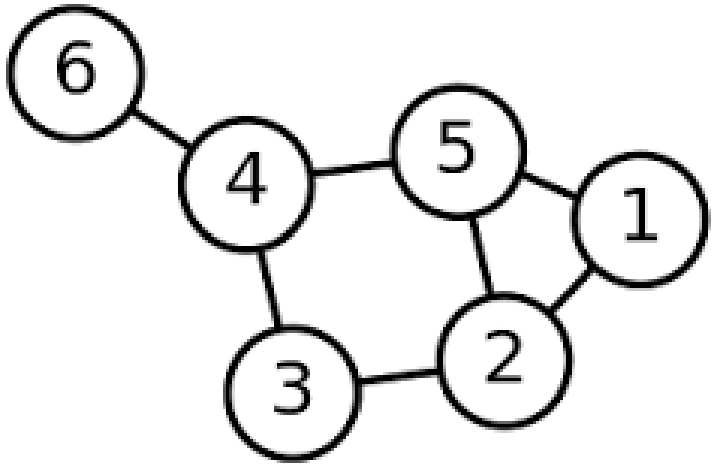



| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | ∞ |
| 1 | 1 | 0 | 1 | 1 |
| 2 | ∞ | ∞ | 0 | 1 |
| 3 | ∞ | ∞ | ∞ | 0 |

# 인접 행렬 표현의 그래프화

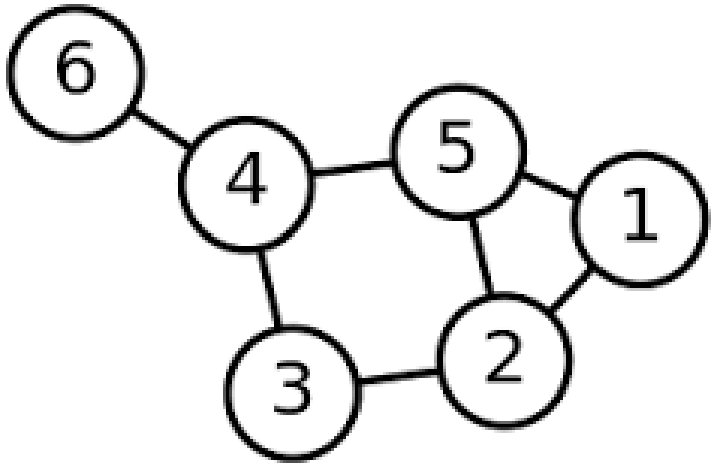
| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | ∞ |
| 1 | 1 | 0 | 1 | 1 |
| 2 | ∞ | ∞ | 0 | 1 |
| 3 | ∞ | ∞ | ∞ | 0 |

# 깊이 우선 탐색 (DFS)




정점 1과 연결된 정점 : 2, 5
정점 2와 연결된 정점 : 1, 3, 5
정점 3과 연결된 정점 : 2, 4
정점 4와 연결된 정점 : 3, 5, 6
정점 5와 연결된 정점 : 1, 2, 4
정점 6과 연결된 정점 : 4

# 넓이 우선 탐색 (BFS)



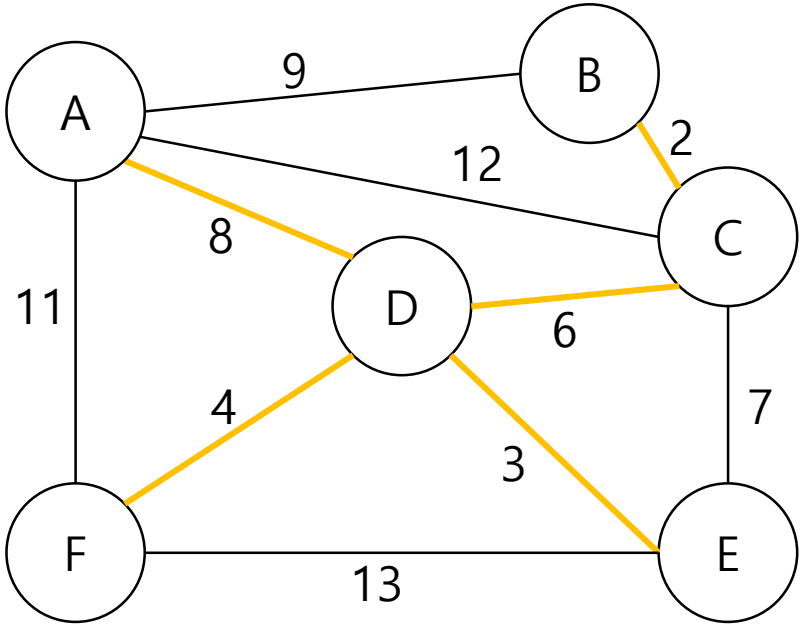

정점 1과 연결된 정점 : 2, 5
정점 2와 연결된 정점 : 1, 3, 5
정점 3과 연결된 정점 : 2, 4
정점 4와 연결된 정점 : 3, 5, 6
정점 5와 연결된 정점 : 1, 2, 4
정점 6과 연결된 정점 : 4

# 크루스칼 알고리즘

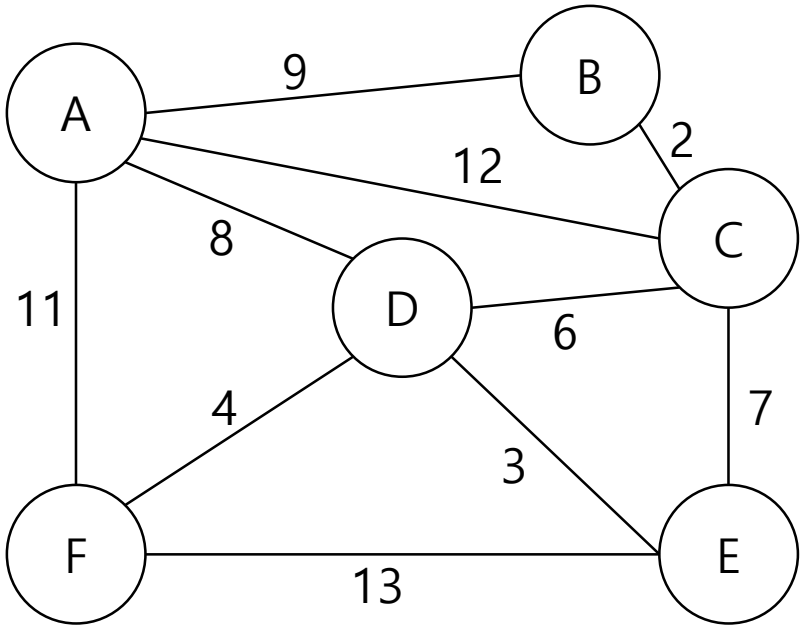
크루스칼 알고리즘 : 가중치를 기준으로 간선을 정렬한 후에
최소 비용 신장 트리를 만드는 알고리즘

최소 비용 신장 트리?

모든 정점을 최소한의 비용으로 돌 수 있는 간선을 모아 만든 트리



# 크루스칼 알고리즘 1

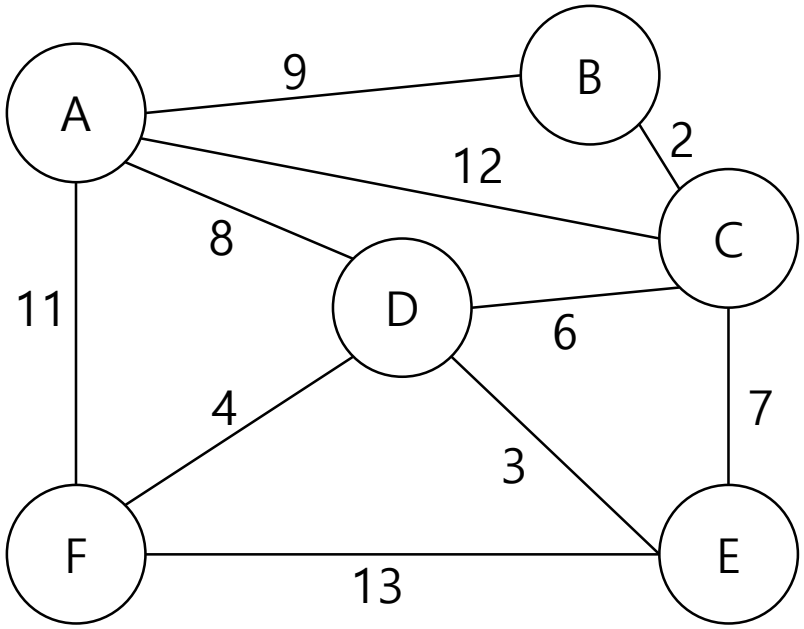



A : A -> B(9) -> C(12) -> D(8) -> F(11)
B : B -> C(2)
C : C -> D(6) -> E(7)
D : E(3) -> F(4)
E : E -> F(13)
F : F

가중치 정렬

| BC(2) | DE(3) | DF(4) | CD(6) | CE(7) | AD(8) | AB(9) | AF(11) | AC(12) | EF(13) |
|---|---|---|---|---|---|---|---|---|---|

# 크루스칼 알고리즘 2

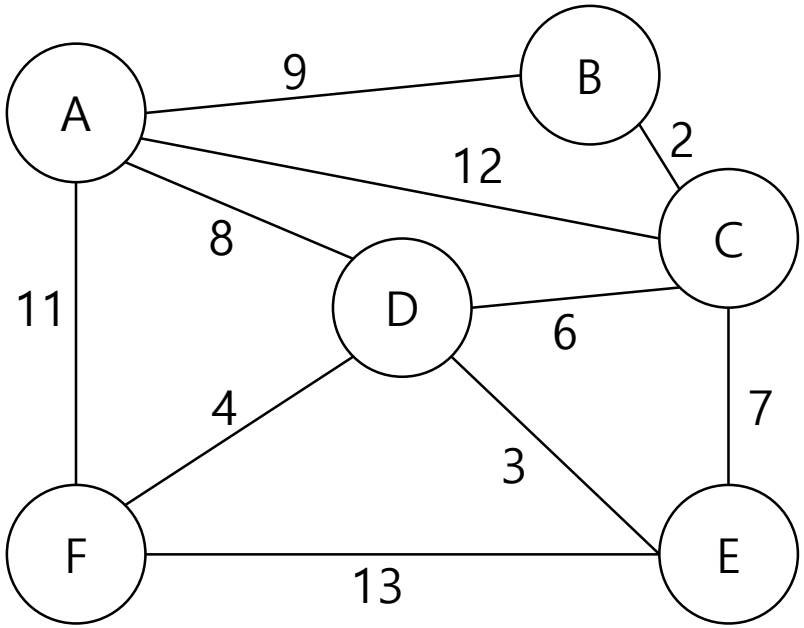



A : A -> B(9) -> C(12) -> D(8) -> F(11)
B : B -> C(2)
C : C -> D(6) -> E(7)
D : E(3) -> F(4)
E : E -> F(13)
F : F

가중치 정렬

| BC(2) | DE(3) | DF(4) | CD(6) | CE(7) | AD(8) | AB(9) | AF(11) | AC(12) | EF(13) |
|---|---|---|---|---|---|---|---|---|---|

# 크루스칼 알고리즘 – 프림 기법




A : A -> B(9) -> C(12) -> D(8) -> F(11)
B : B -> C(2)
C : C -> D(6) -> E(7)
D : E(3) -> F(4)
E : E -> F(13)
F : F

가중치 정렬

| BC(2) | DE(3) | DF(4) | CD(6) | CE(7) | AD(8) | AB(9) | AF(11) | AC(12) | EF(13) |
|---|---|---|---|---|---|---|---|---|---|